Scalable and Declarative Information Extraction in a Parallel Data Analytics System

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

Doktor-Ingenieur
(Dr.-Ing.)

im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
Humboldt-Universität zu Berlin

von
Dipl.-Inf. Astrid Rheinländer

Präsidentin der Humboldt-Universität zu Berlin:
Prof. Dr.-Ing. habil. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:
Prof. Dr. Elmar Kulke

Gutachter:
1. Prof. Dr.-Ing. Ulf Leser
2. Prof. Dr. Felix Naumann
3. Prof. Dr.-Ing. Norbert Ritter

eingereicht am: 24.01.2017
Tag der mündlichen Prüfung: 23.06.2017

Zusammenfassung

Die Menge unstrukturierter Daten ist in den letzten Jahren enorm gewachsen und in diesem Zusammenhang hat sich auch die Analysekomplexität solcher Daten wesentlich erhöht. Informationsextraktion (IE) ist ein bedeutendes Verfahren für viele Anwendungen, in denen unstrukturierte Texte in strukturierte Daten transformiert werden, jedoch erfordert die systematische Anwendung von IE-Techniken auf sehr große Datenmengen hochkomplexe, skalierbare und anpassungsfähige Systeme. Obwohl bereits eine umfangreiche Sammlung von IE-Werkzeugen und Algorithmen für verschiedene IE-Aufgaben existiert, ist die nahtlose und erweiterbare Kombination dieser Werkzeuge in einem skalierbaren end-to-end IE-System immer noch eine große Herausforderung.

Diese Dissertation untersucht genau diese Problemstellung, d.h., es wird ein anfragebasiertes IE-System innerhalb einer parallelen Datenanalyseplattform erforscht und entwickelt, das für konkrete Anwendungsdomänen konfigurierbar ist und für Textsammlungen im Terabyte-Bereich skaliert. Innerhalb dieses Forschungsfeldes werden vier konsekutive Forschungsfragen bearbeitet. Zuerst werden konfigurierbare, algebraische Operatoren für alle grundlegenden IE-Aufgaben und für Web Text Analytics (WA) definiert. Es wird gezeigt wie diese Operatoren genutzt werden können um komplexe IE-Aufgaben in Form von Queries innerhalb der deklarativen Anfragesprache *Meteor* auszudrücken. Solche Queries werden in algebraische *Data Flows* übersetzt, analysiert, logisch und physikalisch optimiert und schließlich in parallele Data Flow-Programme übersetzt, die mit der parallelen Datenanalyseplattform *Stratosphere* ausgeführt werden. Alle Operatoren werden hinsichtlich ihrer physikalischen, algebraischen und Laufzeiteigenschaften charakterisiert um sowohl das Potenzial als auch die Bedeutung der Optimierung der Ausführungsreihenfolge nicht-relationaler, benutzerdefinierter Operatoren für Data Flows (UDFs) hervorzuheben. Als zweite Forschungsfrage wird der Stand der Technik in der Optimierung nicht-relationaler Data Flows untersucht. Relevante Optimierungstechniken, die in verschiedenen Phasen des Optimierungsprozesses in parallelen Datenanalysesystemen eingesetzt werden, werden vorgestellt und existierende Data Flow-Anfragesprachen werden umfassend hinsichtlich der verfügbaren Optimierungstechniken analysiert. Die Analyse kommt zu dem Schluss, dass eine umfassende Optimierung von UDFs für viele Systeme immer noch eine Herausforderung ist. Basierend auf dieser Beobachtung schließt sich die dritte Forschungsfrage an, in der ein erweiterbarer, logischer Optimierer erforscht und entwickelt wird, der die Semantik von UDFs mit in den Optimierungsprozess mit einbezieht (*SOFA*). SOFA analysiert eine kompakte Menge von Eigenschaften, die die Semantik der UDFs beschreiben und kombiniert die automatisierte Analyse mit manuellen UDF-Annotationen, um eine umfassende Optimierung von Data Flows zu ermöglichen. SOFA ist in der Lage, beliebige Data Flows aus unterschiedlichen Anwendungsbereichen logisch zu optimieren, was zu erheblichen Laufzeitverbesserungen im Vergleich mit anderen Techniken führt. Als Viertes wird die Anwendbarkeit des vorgestellten IE-Systems auf realweltliche Textsammlungen im Terabyte-Bereich untersucht, in dem Inhalte des World Wide Webs zu gesundheitsrelevanten Themen mit wissenschaftlichen Veröffentlichungen verglichen werden. Im Rahmen dieser Studie wird systematisch die Skalierbarkeit und Robustheit der eingesetzten Methoden und Werkzeuge untersucht sowie die Qualität der extrahierten Daten analysiert um schließlich die kritischsten Herausforderungen beim Aufbau eines IE-Systems für sehr große Datenmenge zu charakterisieren.

Abstract

In recent years, the size of unstructured data has grown tremendously and the complexity of the analysis of such data has increased significantly. In many domains, information extraction (IE) is an important technique to turn unstructured texts into structured fact databases, but systematically applying IE techniques to very large inputs requires highly complex, adaptable, and scalable systems. Although a number of tools for different IE tasks exist, their seamless, extensible, and scalable combination into a large-scale end-to-end text analytics system still is a true challenge.

This thesis addresses exactly this problem, i.e., we research and develop a query-based IE system that is accurate, configurable towards concrete application domains, and scalable to Terabyte-scale text collections inside a parallel data analytics system. Within this topic, we conduct four consecutive research tasks: First, we introduce a set of domain-independent, algebraic operators, which address all fundamental tasks in IE and web text analytics (WA) and which can be used to express complex IE tasks in form of queries inside the declarative data flow language *Meteor*. Such queries are parsed into algebraic data flows, which are logically and physically optimized, translated into parallel data flow programs, and executed with the parallel processing system *Stratosphere*. We characterize all operators with physical, algebraic, and runtime properties to highlight both the potential and importance of optimizing the execution order non-relational, user-defined data flow operators (UDFs). Second, we survey the state-of-the-art in optimization techniques for data flows with UDFs, which are applied at different stages of the optimization process in parallel data analytics systems. We provide a comprehensive overview on declarative data flow languages for parallel data analytics systems from the perspective of their build-in optimization techniques and conclude that comprehensive optimization of UDFs and non-relational operators still is a true challenge for many systems. Third, we introduce a semantics-aware and extensible logical optimizer for data flows with UDFs based on this observation. Our optimizer builds on a concise set of properties for describing the UDF's semantics and combines automated analysis of UDFs with manual annotations to enable comprehensive data flow optimization. We show that our approach is capable of reordering data flows of arbitrary shape from different application domains, leading to considerable runtime improvements and clearly outperforming plans found by other techniques. Fourth, we study the real-life applicability of our system to Terabyte-scale text collections in a challenging setting to compare the "web view" on health-related topics with that derived from a controlled scientific corpus. We systematically evaluate scalability, quality, and robustness of the employed methods and tools and also pinpoint the most critical challenges in building such a system.

Acknowledgments

After a period of almost six years, today is the day: writing this note of thanks is the finishing touch on my dissertation. It has been a period of intense learning for me, not only scientifically, but also on a personal level and I would like to thank the people who have supported me most during this thime.

First and foremost, I would like to express my sincere gratitude to my advisor Ulf Leser for his continuous support and encouragement during my academic life over the past 10 years. Ever since I was an undergraduate student, he gave me the opportunity to work on amazing projects, which raised my interest in computer science research in general and in large-scale data management in particular. His guidance, patience, and immense knowledge helped me in all this time and especially during researching and writing this thesis. Thank you, Ulf!

I greatly appreaciate the inspiring environment provided by all members of the DFG research group *Stratosphere – Information Management on the Cloud*. Many thanks to all principal investigators and my fellow graduate students, who listened and engaged in many fruitful discussions during our project meetings. Especially Arvid Heise, Fabian Hueske, and Felix Naumann helped a lot to sharpen and improve results presented here by asking the right questions and by challenging me constantly. I am grateful for the dedication of my student assistants Anja Kunkel, Martin Beckmann, and Jörg Meier, who worked tirelessly to ensure that our system kept up to speed with the development of Stratosphere's core components.

I would also like to thank my fellow graduate students and co-workers at WBI, who made sure that I always enjoyed driving to Adlershof in the morning. I thank Stefan Kröger for being the best office mate I could imagine having for more than five years. Especially spending many coffee breaks with Karin Zimmermann, Philippe Thomas, Marc Bux, and Birgit Heene and chatting about life-related topics provided just the right amount of distraction whenever I needed it.

Most importantly, I would like to thank my family for their continuous support and love. Especially Christian, who encouraged me to study computer science in the first place and who always listened patiently to the challenges I faced, greatly contributed to the completion of this thesis.

Thank you very much!

Berlin, 30. June 2017

Contents

1 Introduction

1.1 Motivation

Around 1450, when the Mainz goldsmith Johannes Gutenberg invented a machine-operated printing system with movable metal letters, the methods of book production were revolutionized and caused a media revolution in Europe. Books became mass articles available to many people, which laid the foundations of today's knowledge society and to the development of sciences.

Ever since then and especially since the beginning of the digital revolution in the 20th century, the size of unstructured data (e.g., texts, videos, pictures, etc.) has grown tremendously. The growth of such data within the past 10 years has out-paced the growth of structured data according to the American market research and analysis company IDC [EMC Digital Universe, 2015]. As shown in Figure 1.1, the IDC observed that from 2005 on, the data digitally available has doubled every two years and is estimated to reach $4 * 10^6$ PB (40 zettabytes) in 2020 (excluding sensor data), much of which has an unstructured form. Although costs for hardware capable of storing those data dramatically decreased within the past years and modern data processing systems for large-scale data analytics are available [Bajaber et al., 2016], IDC estimates that only 1–3 % of the available data is analyzed and indexed, indicating that much information is not accessible for complex analyses both in scientific and business scenarios.

Information extraction (IE) is an important technique to turn unstructured texts into structured fact databases and is a fundamental step in various data analysis problems. IE systems often consist of highly complex and domain-specific pipelines of natural language processing (NLP) and IE algorithms including preprocessing steps (such as text segmentation), linguistic analysis (such as sentence parsing, part-of-speech tagging, or stop word removal), which are necessary to enable entities and relationship detection [Sarawagi, 2008; Feldman and Sanger, 2006]. IE has a long tradition in many research communities, for example, in computational linguistics to perform semantic text analysis [Grishman and Sterling, 1990], in business intelligence to identify strategic business opportunities [Raisinghani, 2003; Chaudhuri et al., 2011], or in information retrieval to improve search results [Moens, 2006]. Very often, IE is performed in domain-specific settings, such as biomedicine [Cohen and Hersh, 2005; Thomas et al., 2012], geographical sciences [Wang et al., 2007], or web analytics [Etzioni et al., 2008], which require models adapted to the concrete domain to achieve accurate results in terms of precision and recall. For example, in biomedical IE, extracting information on entities, such as genes, drugs, diseases, or cells, and relationships between those entities requires the use of several heavy-weight tools and algorithms, some of which have a runtime complexity that is quadratic in the text length [Leser and Hakenberg, 2005]. IE is also a prominent topic in database research, where researchers focus mostly on improving scalability and flexibility of the methods by developing declarative query lan-

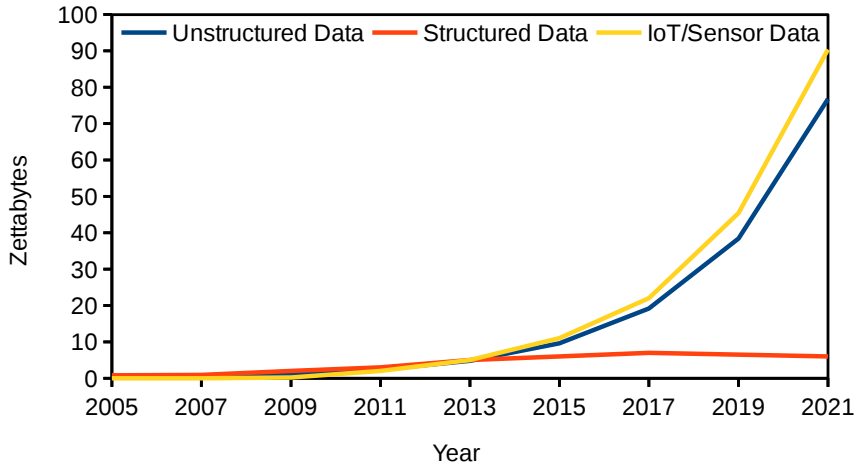


Figure 1.1: Estimated growth of unstructured and structured data according to IDC [EMC Digital Universe, 2015].

guages for non-relational applications at large scale [Shen et al., 2007; Chen et al., 2008; Reiss et al., 2008]. Due to the embarrassingly parallel nature of IE on large document collections, a second line of research to alleviate runtime problems in complex IE programs is to parallelize all analysis using parallel data flow systems on distributed infrastructures such as clusters or clouds [Lin and Dyer, 2010].

Recently, general parallel data analytics systems, most of which generalize the Map/Reduce programming model [Dean and Ghemawat, 2004], have gained much in popularity, as these systems promise to ease writing of scalable programs for analyzing huge amounts of data (e.g., Apache Hadoop [White, 2009], Apache Spark [Zaharia et al., 2010], Stratosphere [Alexandrov et al., 2014], Apache Flink [Carbone et al., 2015]). However, developing data flow programs for analyzing unstructured data sets can become quite time-consuming due to the complexity of the involved tasks. Many query languages for expressing data flows in form of queries or scripts have been developed, for example, Jaql [Beyer et al., 2011], Dremel [Melnik et al., 2010], Pig Latin [Olston et al., 2008], or HiveQL [Thusoo et al., 2009]. These systems often provide only basic operators for simple, SQL-style operations (e.g., aggregations, joins, or filters). Analytic functionality beyond those tasks must be embedded in user-defined functions (UDFs) on a case-by-case basis, where the UDF's semantics is transparent to the query compiler and optimizer. Furthermore, since advanced IE algorithms are often complex, a re-use of existing algorithms and tools is necessary for cost-effective text analysis at large scale [Chiticariu et al., 2010a]. Another key technique to achieve scalability for processing very large document collections is optimizability, since the execution order of IE operators greatly impacts the overall performance of the IE system [Wachsmuth et al., 2011], an aspect, which is not yet addressed properly in parallel data analytics systems. Thus, integrating advanced IE functionality as first-class citizens into query and data flow languages to enable optimizable and configurable IE at large-scale still is a true challenge.

1.2 Goals and contribution

The main goal of this thesis is to develop a query-based IE system inside a parallel data analytics system that is accurate, configurable towards concrete application domains and scalable to large-scale text processing. We aim at enabling users to formulate complex IE tasks in a structured, declarative query language. Such queries are parsed, logically and physically optimized, translated into parallel data flow programs, and finally scheduled and executed on parallel processing engines. Specific contributions of this thesis to the objective of scalable and declarative information extraction on parallel data analytics systems are:

1. We design and implement a query language, data model, and domain-independent operators for information extraction and web analytics (WA), which can be tailored towards different domains and compiled into complex, data flows using the parallel data analytics system Stratosphere [Alexandrov et al., 2014]. We evaluate performance and scalability of all operators in isolation and by executing real-life, complex IE queries in distributed settings.

2. We survey the state-of-the-art in optimizing non-relational data flows, which contain many UDFs, and discuss advantages and limitations of the existing approaches. We present techniques for syntactical data flow modification, approaches for inferring semantics and rewrite options for UDFs, and methods for data flow transformations both on the logical and on the physical level. Furthermore, we provide an overview on declarative data flow languages for parallel data analytics systems from the perspective of their build-in optimization techniques.

3. We research and develop a novel approach for optimizing complex data flows with UDFs, which combines automated analysis of UDFs with manual annotations to enable comprehensive data flow optimization. A salient feature of our approach is extensibility: User-defined operators and their properties are arranged into a subsumption hierarchy, which considerably eases integration and optimization of new UDFs. We evaluate our approach on a selection of data flows that contain UDFs from different domains and compare its performance to three other methods for data flow optimization.

4. We study the real-life applicability of our query language, operator design, and optimization approach in a challenging setting to compare the "web view" on health-related topics with that derived from a controlled scientific corpus. This study combines a focused crawler, applying shallow text analysis and classification to maintain focus, with our text analytics system built inside Stratosphere using a small set of declarative data flows to facilitate web text analytics. We systematically evaluate scalability, quality, and robustness of the employed methods and tools and pinpoint the most critical challenges in building such a system.

1.3 Thesis outline

The remainder of this thesis is structured as follows:
Chapter 2 introduces basic concepts and definitions relevant throughout this thesis. The focus lies on large-scale information extraction and an introduction of the parallel data analytics system Stratosphere, into which we integrated our contributions regarding scalable information extraction.

Chapter 3 presents operators for declarative text analytics that enable parallel information extraction and web analytics. After defining a data model, we introduce elementary and complex operators together with example queries and rewrite options both for WA and IE operators and summarize operator properties relevant to data flow optimization.

Chapter 4 surveys practical techniques for optimizing complex data flows with UDFs and assesses their applicability in parallel data analytics systems. First, syntactical data flow modification is discussed, followed by approaches for analyzing UDF semantics and rewrite options. After surveying data flow transformations on the logical and physical level, this chapter concludes with an overview on declarative data flow languages and a summary of their build-in optimization techniques.

Chapter 5 introduces a novel approach for extensible and semantics-aware optimization of data flows with UDFs, which builds upon a concise set of properties for describing the UDF's semantics. We evaluate our approach on a diverse set of UDF-heavy data flows and compare its performance to three other approaches for data flow optimization. Finally, we show how our optimizer is integrated into the Stratosphere system to enable the end-to-end development, optimization, and execution of data flows with UDFs.

Chapter 6 reports our experiences from building a large-scale, end-to-end IE system with Stratosphere for comparing the "web view" on health-related topics with that derived from a controlled scientific corpus. We evaluate scalability, quality, and robustness of the employed methods and tools and describe encountered challenges during this project together with ideas for their resolution.

Chapter 7 summarizes the findings of this thesis and gives an outlook to future research directions.

1.4 Own prior publications

Some chapters of this thesis are based on previously published peer-reviewed publications.

Chapter 3 describes our contributions to the high-level language Meteor and algebraic layer Sopremo of the Stratosphere system, which was published in [Heise et al., 2012]. The author's roles can be assigned as follows: Heise and Rheinländer designed the Meteor query language and Sopremo algebraic operator packages. Heise implemented the basic system infrastructure of Meteor and Sopremo together with operator packages for relational data processing and for data cleansing. Rheinländer designed, implemented, and tested operator packages for information extraction and web ana-

lytics. Leich critically examined the manuscript and system design. Leser and Naumann supervised the work.

Chapter 4 contains a survey of optimization techniques for complex data flows with UDFs, which was published in Rheinländer et al. [2017]. The contributions of this chapter can be assigned as follows: Rheinländer selected and reviewed all presented techniques. Rheinländer wrote the manuscript, which was revised by Leser and Graefe.

Chapter 5 presents our extensible and semantics-aware optimizer SOFA, which was published previously in [Rheinländer et al., 2015], [Rheinländer et al., 2014], and [Rheinländer et al., 2013]. The author's roles can be assigned as follows: Rheinländer designed, implemented, tested, and evaluated the data flow optimizer. Operator and property taxonomies as well as rewrite templates were also designed and implemented by Rheinländer. Heise provided the basic Sopremo operator algebra, where the optimizer was implemented, and also provided data cleansing operators for evaluation. Hueske designed read/write set analysis for Map/Reduce-style operators [Hueske et al., 2012], which was adapted to our optimizer by Rheinländer. Kunkel, Stoltmann, and Beckmann implemented a web-based graphical user interface for the optimizer to enable end-to-end system demonstrations under close supervision and based on the specifications provided by Rheinländer. Leser and Naumann supervised the project. Rheinländer drafted the manuscripts [Rheinländer et al., 2015] and [Rheinländer et al., 2013], which were critically revised by Heise, Hueske, Naumann, and Leser. The manuscript [Rheinländer et al., 2014] was drafted by Rheinländer and revised by Leser.

Chapter 6 presents a large-scale study from comparing health-related web pages with scientific publications, which was published in [Rheinländer et al., 2016]. The author's roles can be assigned as follows: Rheinländer designed the study, implemented and tested the analytical data flows, and evaluated extraction results. Lehmann provided the initial framework for distributed focused crawling, which was adapted and extended to the biomedical domain by Rheinländer. Kunkel and Meier implemented operators for boilerplate detection and repairing HTML markup under close supervision by Rheinländer. Leser supervised the work. Rheinländer wrote the manuscript, which was critically revised by Leser.

This thesis was created in the context of the collaborative research unit *Stratosphere – Information Management on the Cloud*[1], which is carried out jointly by the Database Systems and Information Management Group (head: Prof. Volker Markl) and the Distributed Systems Group (head: Prof. Odej Kao) at TU Berlin, the Knowledge Management in Bioinformatics Group (head: Prof. Ulf Leser) and the Database and Information Systems Group (head: Prof. Johann-Christoph Freytag) at HU Berlin, and the Database and Information Systems Group (head: Prof. Felix Naumann) at HPI Potsdam. Since 2011, the Stratosphere research group develops a parallel and adaptive system for complex, large-scale information management of (semi-)structured and unstructured data on massively parallel computing infrastructures, i.e., the Stratosphere system [Alexandrov et al., 2014].

Several works of colleagues conducted in the context of Stratosphere have an impact on this thesis. Arvid Heise lead the development of the algebraic layer Sopremo, the query language Meteor, and operators for declarative data cleansing [Heise, 2015], Stephan Ewen [Ewen, 2014] and Fabian Hueske [Hueske, 2015] lead the development

[1] `http://www.stratosphere.eu`, last accessed:2016-12-15

of the PACT programming model and the physical optimizer of parallel data flow programs, and Daniel Warneke [Warneke, 2011] lead the development of the parallel execution engine Nephele, while I lead the development of the logical optimizer SOFA and operators for declarative text analytics.

2 Fundamentals

In this chapter, we introduce basic terminology and definitions used in the remainder of this thesis. We summarize key tasks and challenges in information extraction on large document collections and we introduce the parallel data analytics system Stratosphere, which we use later for implementing declarative operators for information extraction and web analytics as well as for our contribution to the optimization of complex analytical data flows.

2.1 Terminology

All operators and data flows described in this thesis process semi-structured *records* based on the JSON data model [Bray, 2014]. JSON records can represent six different data types, four of which are atomic (i.e., strings, numbers, booleans, and null) and two of which are structured (i.e., objects and arrays). A JSON object is an unordered collection of pairs, each consisting of an attribute name and an associated value representing one of the JSON data types. A JSON array is an ordered sequence of zero or more values. Listing 2.1 shows an exemplary JSON record describing the content of a book together with meta data and sentence and entity annotations.

Definition 1 (Data set) *Any unordered bag of JSON records, which may be accessed individually, in combination, or as a whole, is called data set.*

Listing 2.1: Exemplary semi-structured record of a book's content.

```
{"book":
 {
  "id": "01",
  "title": "1984",
  "author": "George Orwell",
  "text": "It was a bright cold day in April, and the clocks were striking thirteen. ...",
  "annotations": {
     "sentences": [{"sid": "0", "start": "0", end: "73"}, {...} ],
     "entities": [{eid: "0","start": "28","end": "33","entity": "April","type": "date"},
                  {...} ]
  }
  "publisher": "Secker & Warburg, London"
  "published": "1949-06-06"
 }
}
```

The system we develop in this thesis does not require a precise and closed schema definition in the first place, but single operators might require that processed records adhere to a certain schema. For example, text processing operators might require that a record contains an attribute "text", which contains an atomic string. Whether this requirement is fulfilled or not is checked at the operator's execution time.

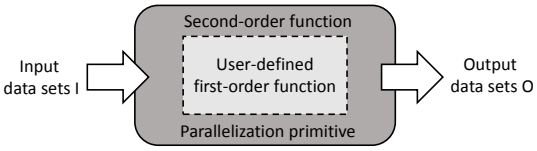


Figure 2.1: Interplay of user-defined first-order functions and parallelization primitives in operators.

Definition 2 (User-defined operator) *An user-defined operator o transforms a list of input data sets $I = [I_1, \ldots, I_n]$ into a list of output data sets $O = [O_1, \ldots, O_n]$ by applying a user-defined function (UDF) f to I.*

Operators can be relational (e.g., selection, projection, join) or non-relational. We use the terms user-defined operator and UDF synonymously to refer to all non-relational operators integrated into parallel data analytics systems by developers or supplied as part of a data flow by users. We also consider relational operators, which are configured with a UDF, as user-defined operators. For example, a join operator, which joins records based on a similarity-based join condition (e.g., Jaccard similarity of two strings), is considered as a user-defined operator. Note that we only consider batch processing in this thesis. All operators require that I is completely given and O is produced by executing atomic operations.

Operators can be either *abstract* or *concrete*. For example, an operator for annotating person names in texts is abstract, and its concrete instantiations are different algorithms and tools for performing this task. Concrete operators may use very different implementations for a given abstract task; for example, the recognition of person names may be performed using dictionaries, patterns, or machine-learning-based methods (cf. Section 2.2).

Concrete operators can either be *elementary* or *complex*. Elementary operators are implemented using a single second-order function, which provides a concrete execution and parallelization semantics as shown in Figure 2.1. Complex operators are composed of multiple elementary operators. They are of high practical relevance, as they provide a shortcut for adding one or more subflows to a data flow. An example of a complex operator for extracting person names from texts is shown in Figure 2.2; details on semantics and implementation of complex operators for IE tasks will be provided in Chapter 3. Complex operators are also highly important for data flow optimization, since a complex operator may exhibit different semantics than its elements as will be discussed in Chapter 5.

Definition 3 (Data flow) *A data flow is a connected directed acyclic graph $D(V,E)$ with the following properties:*

- *vertices $v \in V(D)$ are either operators, data sources, or data sinks,*
- *edges $(v_i, v_j) \in E(D)$ connect operators, data sources, and sinks,*
- *nodes $v \in V(D)$ with an in-degree $deg_{in}(v) \geq 1$ and an out-degree $deg_{out}(v) \geq 1$ are called operators,*

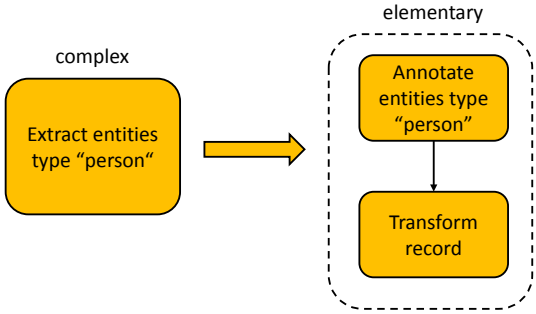


Figure 2.2: Exemplary complex operator for entity extraction and its decomposition into a partial data flow consisting of elementary operators.

- *nodes $v \in V(D)$ with an in-degree $deg_{in}(v) = 0$ and an out-degree $deg_{out}(v) \geq 1$ are called data sources, and*

- *nodes $v \in V(D)$ with an in-degree $deg_{in}(v) \geq 1$ and an out-degree $deg_{out}(v) = 0$ are called data sinks.*

Any induced subgraph $D[V']$ of D with $V' \subset V(D), E = \{(u,v)|u,v \in V' \wedge (u,v) \in E(G)\}$ is called *partial data flow*. We distinguish between *logical* and *physical* data flows. The former is an abstract, algebraic representation of all operations to be performed in a data flow. The latter defines a concrete *execution plan* for a data flow that consists of parallelization functions (e.g. map, reduce) and concrete implementations for each operator as well as data shipment strategies between sources, operators, and sinks for parallel execution. Note that we focus on deterministic, acyclic data flows. Data flows containing iterations or window-based stream processors are out of scope of this thesis (see, e.g., [Hirzel et al., 2014; Ewen, 2014])

By the term *query* we mean a high-level representation of a data flow, which is either formulated in a structured, textual language (e.g., [Beyer et al., 2011; Heise et al., 2012; Olston et al., 2008; Thusoo et al., 2009]), or alternatively, is created by drag and drop of operators in a graphical user interface[2]. We will introduce Meteor, a concrete query language for Stratosphere, in Chapter 2.3.

Definition 4 (Precedence graph) *Two operators o_i, o_j of a data flow D are in a precedence relation, if a path from o_i to o_j in D exists and o_j accesses information contained in attributes $a_n, \ldots, a_m$ that were modified or created by o_i.*

The precedence graph P_D for D is a directed, acyclic graph with the following properties:

- *$V(P_D) = V(D)$ and*

- *$E(P_D) = \{(o_i, o_j)|o_i, o_j \in V(P_D)$ are in a precedence relation$\}$.*

- *Note that $P(D)$ may be disconnected.*

[2]Hadoop User Experience, http://gethue.com/, last accessed: 2016-05-20.

Data sources and data sinks are in a precedence relationship with all downstream (or upstream, respectively) operators to ensure correctness of the data flow during optimization. Given that two operators in D are not in a precedence relation, such *degrees of freedom* enable the optimizer to change the execution order of operators to retrieve the same result more efficiently.

Definition 5 (Semantically equivalent data flows) *Two deterministic data flows D, D' are semantically equivalent (denoted with $D \equiv D'$), if D and D' always produce the same output sets O given the same input datasets I, although intermediate results may differ.*

Finding semantically equivalent data flows is fundamental for any kind of data flow optimization considered in this thesis. By the term *data flow optimization*, we refer to two orthogonal strategies for reducing the total execution costs of a data flow by (1) minimizing time consumption through maximizing the output per time unit for a fixed set of resources or by (2) minimizing the resource consumption necessary to compute the output for a fixed time budget, which can be defined as follows:

Definition 6 (Data flow optimization) *Given a data flow D, a precedence graph P_D, and a cost function $costs$, data flow optimization first determines the set S of semantically equivalent data flows for D, such that $D \equiv D'$ holds for each data flow $D' \in S$. A subset $S' \subseteq S \cup \{D\}$ is called optimal with respect to $costs$ if $\operatorname{argmin} costs(D')$ holds for each $D' \in S'$. In a second step, one data flow $D' \in S'$ is selected for parallel execution.*

However, there are various reasons why an optimized execution plan selected by a data flow optimizer may not be the best possible plan. First, the number of possible plan alternatives for a given data flow may be too large to be considered completely, which is often the case for large data flows with many degrees of freedom. Second, the semantics of UDFs is often not available to the optimizer, which hampers an optimal placement of these operations inside a plan. Third, methods for cost modelling of operators and data transfer are often imprecise. In contrast to relational database settings, where data sets are assumed to be queried repeatedly and a priori computed statistics are available, data flows are usually executed only once and statistics collection on very large data sets may be prohibitively expensive [Cuzzocrea et al., 2011]. Therefore, optimization in parallel data analytics systems is often not carried out cost-based but employs heuristic rules for plan space pruning. To find efficient operator execution orders for data flows in general, different constructive approaches have been proposed which evaluate precedence constraints determined earlier to construct alternative data flows using bottom-up or top-down plan enumeration algorithms [Burge et al., 2005; Hueske et al., 2012; Srivastava et al., 2006].

A *parallel data analytics system* processes analytical workloads on massive data sets in a parallel manner, either by using several parallel threads on a multi-core machine, by executing the workload on different machines in a distributed environment, or by both. In contrast to workloads processed by relational database systems, data flows processed by parallel data analytics systems are usually long-running, ad hoc specified, contain many UDFs, and are executed only once over a certain set of input data [Dean and Ghemawat, 2010].

Queries formulated in data flow languages are typically translated into parallel, executable data flow programs using a compilation process similar to query processing

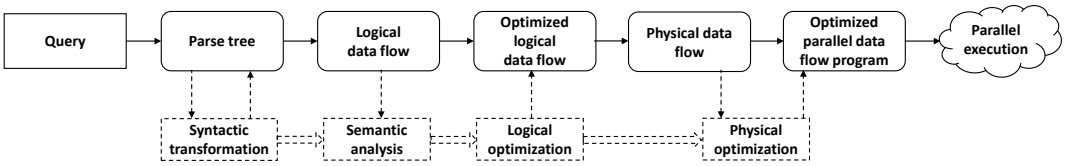


Figure 2.3: Overview of query and data flow processing in parallel data analytics systems.

in relational database systems, see Figure 2.3. A query is translated into an abstract parse tree and *syntactically transformed* in a first optimization step by analyzing the utilization of variables, operators, and predicates. If queries are nested, syntactic transformation attempts to unnest the query to facilitate more comprehensive rewriting in downstream optimization. A data flow consisting of logical operators is created, which is afterwards *logically optimized*, for example by reordering operators, by operator decomposition, or by redundancy elimination. Such optimizations can only be applied if information on operator semantics and the operator's potential for reordering with other operators is available. Operator semantics can be partly determined in a separate optimization step that analyzes *precedence relationships* within a given data flow and its contained operators to infer concrete rewrite options. The optimized logical data flow is translated into a physical data flow and *optimized physically* to reduce both communication and computation costs on the available hardware, for example, by introducing early aggregation and caching into the data flow, or by choosing specific operator implementations and parallelization schemes based on the properties of the data to be processed. Finally, the executable code of the parallel data flow program is created and executed in parallel on the given hardware infrastructure.

In this work, we focus on logical optimization of data flows that contain UDFs. Physical optimization in the context of the Stratosphere system (cf. Chapter 2.3) is studied by Hueske [2015].

2.2 Information extraction

Information extraction (IE) refers to automatically transforming unstructured, natural language text into machine-understandable, structured records relevant for a certain topic or domain [Sarawagi, 2008]. A typical goal is to identify concepts of a certain class in the specific topic or domain while ignoring irrelevant information. Listing 2.2 displays the excerpt from a news article shown in describing an outbreak of Anthrax in Russia in 2016 together with structured records extracted from this article.

Extracting such structured information not only requires to identify semantic units (e.g., sentences, phrases, tokens) in the text, but also to identify the grammatical structure of sentences, the roles of contained tokens in sentences (e.g., nouns denoting persons, geographic locations, or diseases), and to understand syntactic relationships between entities. Moreover, domain-specific background knowledge is necessary to correctly assign extracted information to the given structured representation, i.e., cor-

[3] Source: The Disease Daily, `http://www.healthmap.org/site/diseasedaily/article/anthrax-outbreak-siberia-harbinger-unfreezing-pathogens-81616`, last accessed: 2016-08-31

Listing 2.2: Excerpt from a news article on a disease outbreak in Russia in 2016[3]and extracted records from this article.

```
Anthrax Outbreak in Siberia as a Harbinger of the Unfreezing of Pathogens
On August 1, 2016 in Northern Russia, a 12-year-old boy died of anthrax, marking the
first fatal case among 20 confirmed infected humans from the Yamalo-Nenets region in
Siberia. The anthrax outbreak has taken a heavy toll on the reindeer population,
killing off 2,300 of the population as of August 2nd The regional government has
declared a state of emergency; response has included: quarantine of the area,
evacuation and hospitalization of potentially exposed families, burning of reindeer
remains, and vaccination of healthy reindeer. ...

Type:       Disease outbreak
Disease:    Anthrax
Date:       August 1, 2016
Locaction:  Yamalo-Nenets
Region:     Siberia, Russia
Casualty 1: 12-year-old male, dead
Casualty 2: 19 humans, infected
Casualty 3: 2,300 reindeers, dead
```

rectly identifying infected and deceased people as casualties from a disease outbreak in our example [Chan and Roth, 2010].

2.2.1 Tasks in information extraction

IE is a challenging problem due to the complexity and ambiguity of natural language texts, which often contain homonyms, synonyms, and implicit mentions of relevant facts distributed across different sentences. For example, information on casualties of the Anthrax outbreak in Listing 2.1 is distributed across two sentences and the deceased 12-year-old is included in the calculation of the number of infected people. IE requires multiple steps to create structured information, which are usually executed successively as shown in Figure 2.4. Important tasks are

- text preprocessing,
- text segmentation,
- linguistic analysis, and
- information extraction.

Note that other NLP problems, such as word sense disambiguation, sentiment analysis, or semantic role labeling, relate to IE, but are not considered here. For an overview, we refer the reader to [Manning and Schütze, 1999].

Text preprocessing

Preprocessing is a critical step in any IE system, which is applied to heterogeneous document collections. The largest resource of freely available, unstructured documents is the open web, which provides texts in a plethora of languages, document formats, and character encodings. A fundamental step to ensure effective IE on such document

collections is to prepare and transform the documents into a consistent format and character encoding, which can be processed by IE and NLP tools. Furthermore, most IE and NLP tools are not language-agnostic and require its input to be available in a certain language. The choice of preprocessing methods depends on the concrete application and domain. For example, for processing scanned PDF documents with an IE system, optical character recognition is necessary to transform the scanned images into machine-readable text.

Text segmentation

Text segmentation, i.e., the separation of continuous text into meaningful components (sentences, phrases, and individual tokens) is fundamental for many IE tasks. It is also an important preparation for downstream IE methods such as part-of-speech tagging or sentence parsing, which require upfront information on the beginning and ending of sentences and individual tokens.

Sentence splitting. Often, IE is carried out on the sentence level, since sentences are a basic unit of meaning that group entities and statements. A fundamental task for IE is therefore to detect sentence boundaries in texts, which is not trivial due to the ambiguity of contained punctuation marks. Naïvely searching for delimiters (".", "!","?") cannot accurately split text into sentences, since different language characteristics hamper the detection of correct sentence boundaries, such as:

- abbreviations and proper names (e.g., "There are 9,500 people working for Yahoo!, most of them are located in the U.S."),
- errors introduced during preprocessing (e.g., "2016 Alzheimer's disease facts and figures.Abstract.This report describes . . . "),
- the existence of direct speech (e.g., "'Today's lecture is on Big Data analytics', she said."), or
- contained technical content (e.g., "A free search engine for Medline is available at the website of Pubmed (`http://www.ncbi.nlm.nih.gov/pubmed/`).").

Different approaches have been developed by the research community and are available in open-source IE and NLP toolkits, such as sentence splitting based on logistic regression (OpenNLP [Baldridge, 2005]), regular expressions and rules (GATE [Cunningham, 2002]). A two-phase approach, which first tokenizes text using finite state automata and subsequently detects sentence boundaries is employed in the Stanford NLP toolkit [The Stanford Natural Language Processing Group, 2016]. Sentence splitting based on machine-learning yields the highest accuracy [Tomanek et al., 2007], but is also the most compute-intensive method.

Tokenization. Given a string, the task of tokenization is to segment this string into individual pieces (*tokens*) by removing certain whitespace characters or word delimiters. Existing tokenizers for languages based on the Latin alphabet often follow heuristics, which consider all contiguous alphanumeric strings as one token. Punctuation, white spaces, or parentheses are accordingly not considered as individual tokens. For ordinary language texts (like newspapers), simple heuristics suffice to achieve reasonable accuracy of the tokenizer, however, for languages without token boundaries such as Chinese or Thai, more complex heuristics or sophisticated language models are needed.

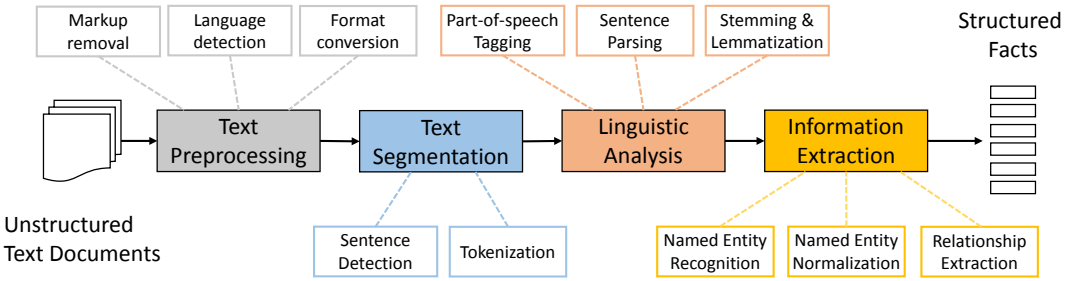


Figure 2.4: Pipelined architecture of information extraction processes.

Linguistic analysis

In contrast to humans, which can easily derive meaning from spoken or written language, computers need explicit annotations of the sentence structure to extract relevant information accurately. Thus, analyzing texts linguistically, i.e., understanding the morphological and syntactical structure of tokenized sentences, is crucial [Manning and Schütze, 1999]. Morphology studies the structure of words (stems, prefixes, infixes, suffixes) and identifies their part of speech, i.e., assigning words with similar grammatical properties to lexical categories such as nouns, verbs, adjectives, etc. The syntactic structure of sentences is represented by parse trees, which are built by analyzing dependencies of contained words or by analyzing the sentences' phrase structure. In linguistics, a distinction is also being made between open and closed grammatical word classes. In open word classes, new words can be added to the class whenever necessary, whereas closed word classes consist of a finite set of words, which is not or only very rarely expanded over time.

Part-Of-Speech Tagging. The grammatical class of an individual token is called *part-of-speech (POS)*. Assigning part-of-speech tags to tokens is helpful for many tasks and especially important for named entity recognition, since named entities should belong to the noun class. POS taggers are available for different languages (e.g., [Schmid, 1995; Li, 2011; Chanod and Tapanainen, 1995]) and for domain-specific applications [Smith et al., 2004]. POS taggers are often implemented using Hidden Markov Models or based on Maximum Entropy Models [Güngör, 2010].

Stemming and lemmatization. The goal of stemming and lemmatization is to fuse morphological variants of individual words into a defined base form. While stemmers aim at identifying word stems without analyzing morphological properties in depth, lemmatizers analyze the meaning and intended part-of-speech of words in a sentence. Stemming heuristically removes inflectional suffixes, such as plurals ("house" vs. "houses") or present and past participles (e.g., suffixes -ing" and "-ed"). More advanced methods remove derivational suffixes (e.g., suffixes "-ment", "-ize") and also try to avoid stemming of proper nouns (e.g. "Rowling", "Alfred") and apply quantitative restrictions (e.g., remove suffix if resulting stem contains at least three letters –> do not stem "king"). Since stemming ignores semantical meaning of words, it tends to introduce errors to IE processes by over- or under-stemming of words [Paice, 1994]. Therefore, the use of derivational lexicons has been suggested to derive more appropriate base forms [Krovetz, 1993].

Sentence parsing. Sentence parsing refers to the automated analysis to derive the syntactic representation (i.e., parse trees) of a given sentence. The two most common representations are *constituent parse trees* and *dependency parse trees* [Carroll et al., 1998]. Constituent parse trees, which model the phrase structure of a sentence, are built by recursively decomposing a given sentence into smaller units, which are subsequently classified by their internal structure into phrases (noun phrases, verb phrases, etc.). Dependency parse trees are built by analyzing the words of a sentence regarding their relationships to other words (so-called dependencies) and by classification of words regarding their functional role in a sentence (e.g., subject, predicate, object). Sentence parsing is considered as one of the most challenging tasks in NLP due to the ambiguity of language and is often performed by means of statistical inference from samples of annotated language data to disambiguate word meanings [Nivre, 2010].

Named Entity Recognition and Normalization

Named Entity Recognition (NER) detects predefined, domain-specific concepts (e.g., persons, locations, genes, diseases) from unstructured texts and transforms them into structured representations. Next to identifying the concept itself, NER occasionally also includes the detection of additional information, which describe the named entity in more detail. For example, locations might be specified further with a zip code, geographic coordinates, state, or province. NER also involves normalization of named entities (NEN), which maps named entities to canonical identifiers. NEN is especially important for inflective languages and for many open entity classes due to the ambiguity of naming conventions. For example, the breast cancer related gene "BRCA2" has different synonyms, such as "breast cancer 2", "FACD", "FANCD", or "Fanconi anemia, complementation group D1". Since entity names are highly ambiguous and might span multiple, not necessarily contiguous tokens (as the BRCA2 example highlights), NER and NEN are non-trivial tasks and concrete methods to perform NER and NEN are highly application specific [Sarawagi, 2008]. The three fundamental techniques to perform NER and NEN are:

- **Dictionary-based** NER builds upon a dictionary for the entity type of interest, which contains synonyms, spelling variations, and often a canonical ID for each entity. In many cases, dictionary-based NER yields high precision at a rather low recall, since dictionaries are inevitably incomplete for open word classes. Moreover, dictionary-based approaches do not consider the context of matched entities, which may lead to a significant number of false-positive matches. For closed word classes, however, dictionary-based NER provides satisfactory results [Nadeau and Sekine, 2007].
- **Rule-based** NER builds upon a set of rules, which aims at capturing all possible cases and conditions where named entities occur in texts. Rules can either be assembled manually or learned from training data and combine different text and domain properties (e.g., capitalization, part-of-speech tags, trigger words) with dictionary look-ups. Rule-based NER has shown to yield accurate extraction results for different domains [Chiticariu et al., 2010b].
- **Classification-based** NER classifies each token in a text as being (a part of) an entity or not based on a previously learnt model from annotated training data,

which contain positive and negative examples of the named entity class of interest. In current systems, both surface and context features are used for classification, for example, the word itself, character n-grams, or part-of-speech tags of the word and its surroundings. A major advantage of classification-based approaches is that they enable the recognition of yet unseen entities and thus achieve high-quality extraction result at current NER challenges [Krallinger et al., 2013]. At the downside, state-of-the-art classification-based NER needs large sets of training data and the extraction performance in terms of speed is often inferior compared to rule- and dictionary-based approaches.

Relationship extraction

Relationship extraction (RE) identifies relationships between tuples (mostly pairs) of named entities. RE has many applications, for example, to create and augment structured knowledge bases or to support question answering [Sarawagi, 2008]. The three predominant approaches for RE are:

- **Co-occurrence-based** RE assumes that entities, which occur together in the same textual context (e.g., sentence, paragraph, n gram), are related to each other. In general co-occurrence-based RE yields a high recall at a low precision, since RE based on co-occurrences predicts a relationship for every pair of entities within the same context. Recall even increases with the size of the textual context. Co-occurrence-based RE can be performed independent from concrete entity and relationship types, since no sophisticated linguistic analysis of the textual contexts and no training data is required. It is therefore easily adaptable to new application domains and scales well to IE at large scale [Ding et al., 2002].
- **Pattern-based** RE was first introduced by Hearst [1992], who defined patterns for the detection of isA-relationships of different entities. Later, this idea has been absorbed by many others to facilitate RE in different domains, for example [Suchanek, 2014; Banko et al., 2007; Auger and Barrière, 2008]. Similar to pattern-based NER, pattern-based RE requires to first identify and characterize the semantic relation to be detected, to discover concrete patterns from a set of annotated training data, and to search for concrete instances of the pattern in the texts to be analyzed. Rule-based RE achieves high precision at a rather low recall, balancing precision and recall requires many patterns, which are most often defined manually for each relationship type of interest.
- **Classification-based** RE classifies each pair of entities occurring in the same semantic context whether this pair is in a relationship or not. Similar to classification-based NER, annotated training data containing negative and positive examples of relationships is required. To this end, a feature vector for each positive and negative pair of entities is created, which might consist of a diverse set of features, such as part-of-speech tags, distance between entities, length and type of the path in a dependency parse tree, etc. Research has shown that across different domains, classification-based RE often achieves a superior performance compared to pattern- and co-occurrence-based RE [Sarawagi, 2008] and especially SVM-based classification has shown to yield excellent extraction accuracy for RE in complex domains, such as biomedical RE [Irsoy et al., 2012].

2.2.2 Information extraction at large scale

Traditionally, the NLP community is mostly concerned with increasing the precision and recall of the developed IE methods for different application domains. Over the past decade, quite a few systems have been developed, which bundle collections of IE and NLP algorithms to provide functionality for solving fundamental IE and NLP tasks, such as sentence splitting, tokenization, or part-of-speech tagging. The most prominent among them are UIMA [Ferrucci and Lally, 2004], openNLP [Baldridge, 2005], LingPipe [Baldwin and Carpenter, 2003], NLTK [Loper and Bird, 2002], and GATE [Cunningham, 2002]. While openNLP, LingPipe, and NLTK provide libraries and application programming interfaces (APIs) for the available algorithms, UIMA and GATE are comprehensive text processing suites, which also contain execution engines and graphical user interfaces (GUIs) to support end users with creating and executing text processing tasks [Kano et al., 2010]. Most of these systems – except of UIMA (see below) – target IE on small to mid-sized data sets. Efficiency and scalability of these methods, however, has been ignored to a large extend.

Scalability for IE tasks comprises three different dimensions [Agichtein and Sarawagi, 2006]. First, when applying IE to very large corpora, the efficiency of the applied IE tools may not be sufficient for the large text collection. Second, large text collections such as the open web are usually highly diverse due to non-standardized publishing processes, heterogeneous naming conventions, and a large amount of diverse publishing sources. Comprehensively extracting information from such data sets requires many specialized extraction methods (rules, patterns, dictionaries), which need to be adapted to the concrete IE tasks and document collections at hand. Third, domain diversity increases with large text collections, therefore, many specialized rules, patterns, and models covering this diversity need to be developed and maintained.

Early approaches to deal with IE at large scale build on scanning, i.e., all documents are processed until a target recall is reached, or filtering and classifying the document collection to analyze only relevant documents and avoid processing of documents considered irrelevant [Grishman et al., 2002; Ipeirotis et al., 2007; Pantel et al., 2004]. Other approaches exploit index structures to retrieve only relevant documents [Cafarella and Etzioni, 2005; Etzioni et al., 2004; Agichtein and Gravano, 2003]. However, all of these methods only processed mid-sized document collections of at most a couple of 100,000 documents with a size of a few Gigabytes. In this thesis, we are interested in processing document collections of Millions of texts spanning to Terabyte-sized data sets.

In recent years, IE by means of parallel and distributed data processing has gained much attention [Chandramouli et al., 2012],[Khuc et al., 2012],[Furche et al., 2014] due to the ever-increasing sizes of document collections. UIMA Asynchronous Scaleout (UIMA-AS) [The Apache Software Foundation, 2012] is an extension to UIMA, which focuses on increasing the scalability of UIMA to large document collections. UIMA Analysis Engines (AE) are encapsulated as services and can be executed locally or in distributed environments. Communication in UIMA-AS with AEs is carried out asynchronously based on shared queues, i.e., different IE analyses can be requested at the same time before results are returned. Degrees of parallelism can be adjusted for each AE in combined AE pipelines, however, optimization of IE programs through reordering or bottleneck detection is not addressed.

Egner et al. [2007] present UIMA-Grid, which enables parallel and distributed IE analyses using the grid management system Condor [Thain et al., 2005]. In this setting, one or more grid nodes perform document management and preprocessing, such as language detection, tokenization, or document indexing. Subsequently, UIMA programs and subsets of the documents to be analyzed are grouped into jobs and distributed on the grid infrastructure. Optimization or bottleneck detection is not addressed, only a manual decomposition of the UIMA workflows into subunits by the user is supported.

Behemoth [Nioche, 2012] is a system for large-scale document processing in cluster or cloud environments based on Apache Hadoop [White, 2009]. Document processing is carried out through custom wrappers for IE, NLP, and machine learning operations from UIMA, GATE, and Apache Mahout. Combining operations from UIMA, GATE, and Mahout in a single text analytics pipeline, however, is not possible. In Behemoth, documents are first converted into an internal format for processing, IE and NLP data analytics pipelines are specified for one of the available IE and NLP systems, and eventually submitted for parallel execution with Hadoop. Internally, Behemoth converts the data analytics pipelines into Hadoop jobs consisting only of *Map* operations and transparently distributes and executes the jobs in parallel on the given hardware infrastructure. Optimization, for example by task reordering, bottleneck detection, or adjusting degrees of parallelism for individual operations is not supported.

GATE cloud services [Tablan et al., 2013] is an extension to GATE, which provides a distributed, parallel execution of IE programs over document collections utilizing the Amazon EC2 services. IE programs are parallelized only as a whole and optimization by task reordering, bottleneck detection, or varying the degree of parallelism for individual operations is not supported.

System T [Chiticariu et al., 2010a] is a declarative system for large-scale IE based on database technology, where the description of IE tasks and their execution are strictly separated. Information extraction pipelines are written in a declarative, rule-based language called AQL, which shows similarities to SQL. AQL rules are translated into algebraic execution plans, which are optimized cost-based using a handful of rewrite rules specific to IE [Reiss et al., 2008], and executed by the underlying execution engine. Compiled AQL plans can also be executed on parallel and distributed infrastructures using the BigInsights system, a distribution of Hadoop, by wrapping the compiled plans and the System T runtime into JAQL [Beyer et al., 2011] functions, which are translated into Map/Reduce programs. Optimization of AQL plans is carried out for single-threaded execution and does not consider the parallel execution environment.

2.2.3 Problem statement

Although the above-mentioned systems provide a wide range of IE and NLP functionality and mechanisms to scale to large document collections, optimization and extensibility with custom UDFs in a user-friendly manner is not addressed. Moreover, all of the above mentioned systems focus on IE only. Complex analytics, where operations from different application (e.g., web analytics, graph processing, data cleansing) areas must be combined with IE operations, are not supported. Therefore, a central goal of this thesis is to develop a system, which

1. enables the expression of complex IE tasks on parallel data analytics systems in a user-friendly manner through a declarative data flow language,
2. optimizes complex IE data flows comprehensively to scale to Terabyte-sized document collections, and
3. is adaptable to different application domains of diverging complexity.

Before introducing concrete IE operators in Chapter 3, we first give an overview on the parallel data analytics system Stratosphere, which provides the foundation for the design, implementation, and optimization of complex IE data flows studied in this thesis.

2.3 The Stratosphere data analytics system

Stratosphere [Alexandrov et al., 2014] is a full-fledged system for massively parallel data analytics of huge data sets using data flows that contain UDFs. It is jointly researched and developed by the DFG-funded research group *Stratosphere – Information Management on the Cloud*[4] and provides the foundations for the open-source Apache top-level project *Flink* [Carbone et al., 2015]. Stratosphere enables parallel batch- and iterative data flow processing, whereas Flink also supports stream-based processing of huge data sets. In the following, we describe the architecture of Stratosphere for batch-processing with a special focus on its high-level language and the underlying algebraic layer.

2.3.1 System architecture

The architecture of Stratosphere consists of three layers as displayed in Figure 2.5, namely

- *Meteor/Sopremo*, a declarative scripting language and algebraic operator model,
- the physical *PACT* programming model, and
- the parallel execution engine *Nephele*.

Each layer is equipped with its own programming model and specific components responsible for different tasks during data flow processing, which will be described below in more detail.

2.3.2 Meteor/Sopremo: data flow language and operator model

Meteor [Heise et al., 2012] is a data flow oriented declarative scripting language that resides at the top of the Stratosphere stack. Meteor builds upon a semi-structured data model that extends JSON (cf. Chapter 2.1). It has similar objectives as other data flow languages (e.g., Pig [Olston et al., 2008] or Jaql [Beyer et al., 2011]), namely providing a high-level, easy-to-use interface to complex, user-defined operations in data analytics systems to end users. In contrast to other languages, Meteor is based upon the semantically rich and extensible operator model Sopremo, which enables that the

[4] `http://www.stratosphere.eu`, last accessed: 2016-12-14

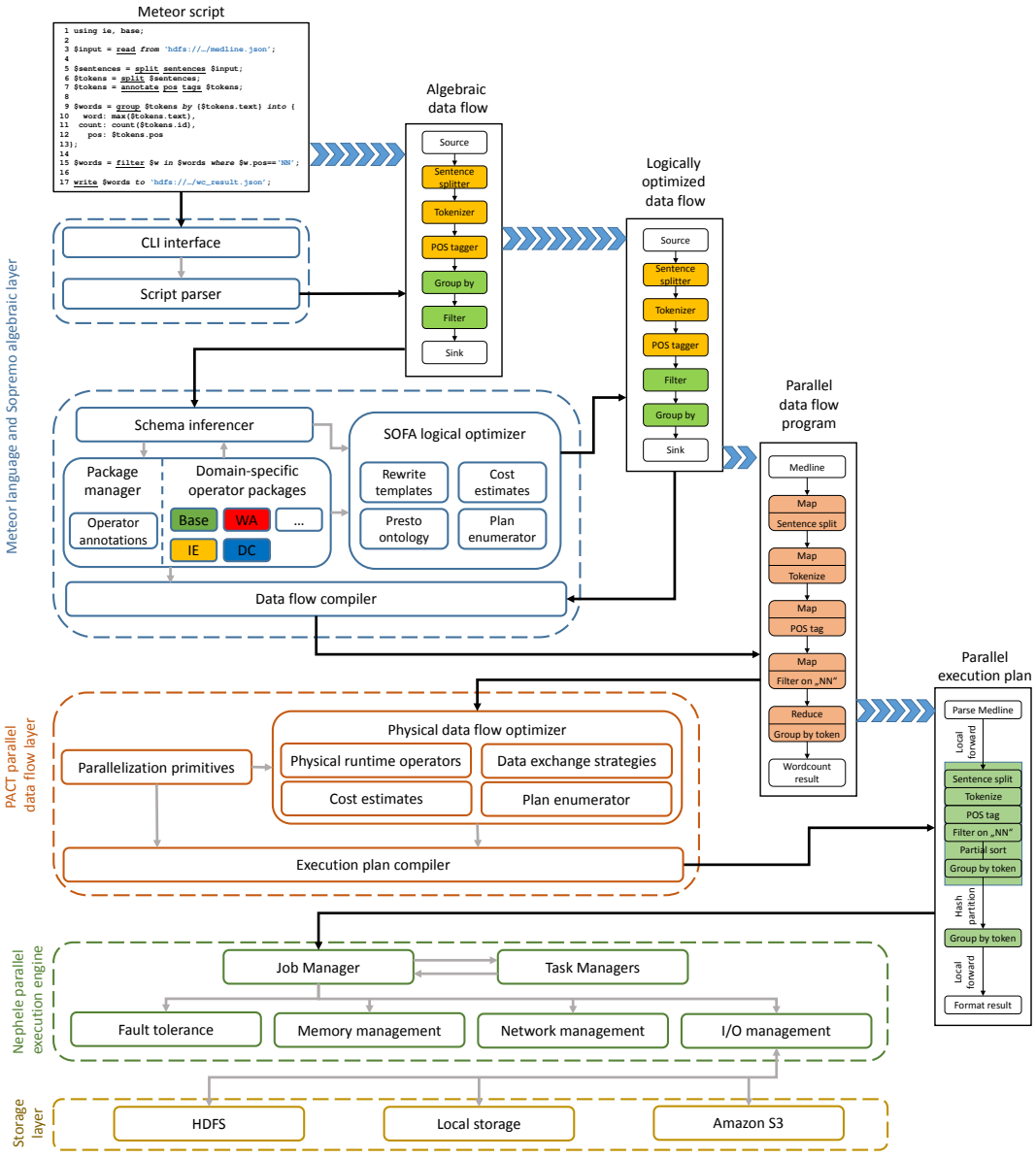


Figure 2.5: Architectural overview and query compilation in the Stratosphere system for parallel data analytics. Declarative Meteor queries are submitted through a command line interface, parsed, and translated into logical data flows in the Sopremo algebra. A schema inferencer analyzes each logical plan to generate a global record schema, which is used by the SOFA optimizer during logical optimization. An optimized logical plan is translated by the data flow compiler into a PACT program, which is physically optimized and eventually executed in parallel by the parallel execution engine Nephele.

operator's semantics can be accessed at compile time and is thus available during data flow optimization.

In Meteor, each operator invocation starts with the unique name of the operator and is usually followed by a list of inputs and a set of operator properties, which are configured with a list of property name and expression pairs. The result of an operator invocation can be assigned to a variable, which either refers to a materialized data set or to a logical intermediate data set. Variables start with a dollar sign ($) to ease distinction between data sets and operator definitions.

An example of a Meteor query, which computes the frequencies of nouns in a collection of documents, is shown in the left part of Figure 2.6. The first line of the script imports Sopremo operator packages (see below), which are used in the query. Here, operator packages for IE and basic operations are used. Line 3 specifies the data source, which in this case is a JSON file stored in the distributed file system HDFS [Borthakur, 2008]. Lines 5–7 apply three different IE operators, which process unstructured text linguistically by first splitting it into distinct sentences (Line 5), by annotating token boundaries (Line 6), and by annotating part of speech tags (Line 7). Lines 9–13 describe an aggregation, which determines the frequencies of tokens inside a `group by` operator. In this operator, the grouping key is specified by the individual tokens (indicated through the operator property *by*) and a `count` function determines the frequencies for each group. The operator property *into* specifies the output schema of the processed and aggregated records, which are subsequently filtered for records identified as a noun (Line 15). Finally, the result set is written to HDFS in Line 17.

Meteor queries are submitted to the system through a command line interface, parsed into abstract syntax trees composed of basic or complex Sopremo operators and translated into logical execution plans in the Sopremo algebra (see next paragraph). The schema inference component analyzes each logical plan to generate a global record schema, which is used by the logical optimizer during optimization and the data flow compiler to translate the logical data flow into a PACT program.

Sopremo operators and packages

Meteor queries are translated one-to-one into data flows consisting of Sopremo operators. Each invocation of an operator in Meteor corresponds to an elementary or complex Sopremo operator and variables in Meteor are translated into data flow edges indicating the data flow between Sopremo operators.

The right part of Figure 2.6 displays the corresponding Sopremo data flow of our example query for noun frequency computation. The data flow is linear and consists of five operators, three of which are elementary operators (i.e., POS tagger, Group by, Filter), and two of which are complex operators (i.e., Sentence splitter, Tokenizer).

All Sopremo operators are organized in domain-specific packages, which are self-contained libraries of the operator implementations, their syntax, and semantic annotations. All operators are either elementary or complex and may have different instantiations (cf. Chapter 2.1). Stratosphere contains four packages, namely a *Base* package containing 16 operators, a package for *IE* with 21 operators, a package for data cleansing (*DC*) with nine operators, and a package for web analytics (*WA*) with five operators. A detailed description of the Base and DC packages is available in [Heise, 2015], details regarding IE and WA operator packages shall be presented in Chapter 3.

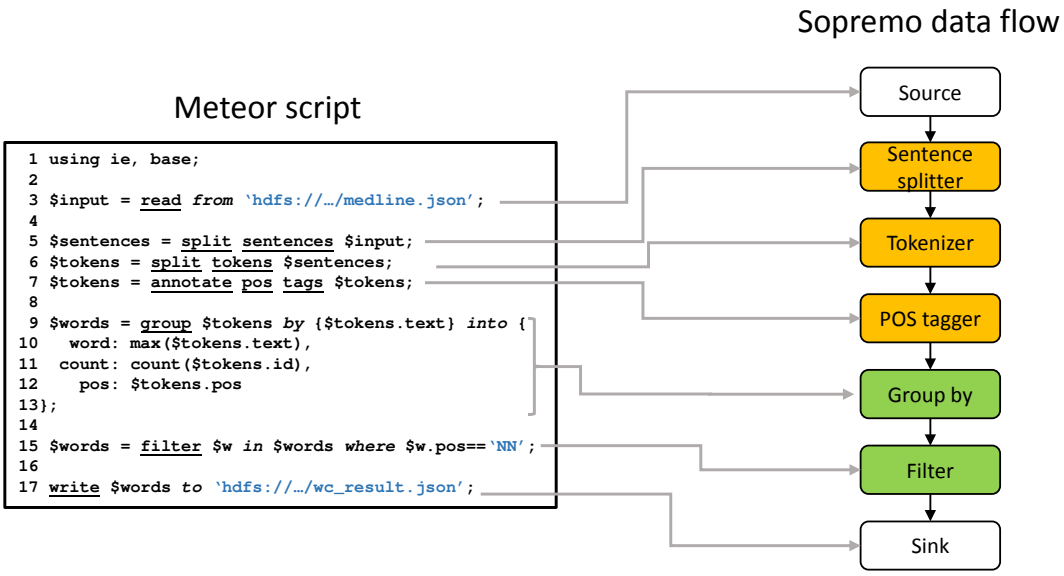


Figure 2.6: Example Meteor query for term frequency computation with corresponding Sopremo data flow. The precedence graph and an logically optimized Sopremo data flow are contained in Figure 2.7

.

The Base package comprises mostly relational operators, such as filter, projection, transformation, join, and group. These operators are complemented by operators for semi-structured data, such as nest or unnest.

The DC package comprises six different classes of operators for data cleansing and data integration [Heise, 2015], which address common challenges of dirty or heterogeneous data sources, such as inconsistent representation of equivalent values, fuzzy duplicates, typographic errors, or missing values.

The IE package comprises three classes of operators: one class for producing text annotations, one class for auxiliary operators to transform or merge annotations, and one class for complex operators. Operators analyze the text and add, remove, or update annotations to the record. They may also transform records, for example, the operator for sentence splitting takes as input single records formed of documents and outputs a set of records formed of sentences.

The WA package comprises operators for analyzing web documents, namely operators for HTML markup repair, for markup removal, as well as operators for extracting links, tables, and other structured information contained in web documents.

Extensibility

By extending the set of existing Sopremo operators, domain experts can easily integrate domain-specific, user-defined operators into the Stratosphere system. To support reuse of available operator implementations in Stratosphere, existing Sopremo operators can be composed, i.e., a set of elementary operators can be interconnected in a partial data flow to form a complex operator. This enables code re-use and the optimization of

complex operators, since rewriting a complex operator may be enabled by transforming its building blocks (see Chapter 5 for details).

Developers of new operators or operator packages need to ensure that all operators are self-contained regarding the following aspects for a seamless integration into Sopremo [Heise et al., 2012]:

Self-contained operator implementations. Package developers provide operator semantics together with parallel operator implementations. New operators can either be defined as elementary operators, each providing it's own low-level parallel implementation (PACTs, see below), or as a complex operators, which are combination of existing operators.

Self-contained operator properties. Properties of Sopremo operators (e.g., join conditions, entity classes to be annotated) are available through a reflection API, which manages and validates requested properties directly within the concrete operator. Properties are used to configure operators and to choose a concrete implementation during optimization.

Self-contained operator annotations. Optionally, package developers can provide annotations, such as cost estimates, specific rewrite rules, or algebraic properties (e.g., commutativity, associativity) as an aid to the logical optimizer inside their package. If such annotations are not provided, the optimizer relies on information inferred from the operator's implementation (see Chapter 5 for details).

Logical data flow optimization

Once a Meteor query was compiled into an algebraic Sopremo data flow and the corresponding global schema was determined, logical data flow optimization through the *SOFA* optimizer commences to determine a semantically equivalent, yet more cost-effective logical plan. In this section, we give a brief high-level overview on SOFA and its components, details of the optimization approach are contained in Chapter 5.

IE data flows are naturally UDF-heavy. A major issue in optimizing such flows is the diversity of the contained UDFs. Defining rewrite rules that respect the individual operator semantics for each possible combination of operators is merely impossible in UDF-rich systems such as Stratosphere. A particular challenge during optimization is extensibility, as every new operator in principle needs to be analyzed with respect to all existing operators to identify rewrite options.

SOFA solves this problem by means of Presto, an extensible taxonomy of operators, properties, and rewrite templates, and by reasoning along subsumption relationships encoded in Presto. The principal ingredients of Presto are two taxonomies describing generalization-specialization relationships (*isA*) both between pairs of operators and pairs of properties. Leaves in the operator taxonomy describe concrete implementations of the abstract parent operator, for example, different concrete algorithms for entity extraction are represented as leaves, whereas the abstract operator for entity extraction is an ancestor of those leaves. Presto uses three additional relationships (*hasProperty*, *hasPart*, and *hasPrerequisite*) to model relations between operators and properties. Properties relevant for optimization are, for example, algebraic properties such as commutativity or associativity, the parallelization function (e.g., map, reduce), or the read/write behavior of operators at attribute level. Rewrite templates are defined using Presto relationships, operator properties, and abstract operators as building

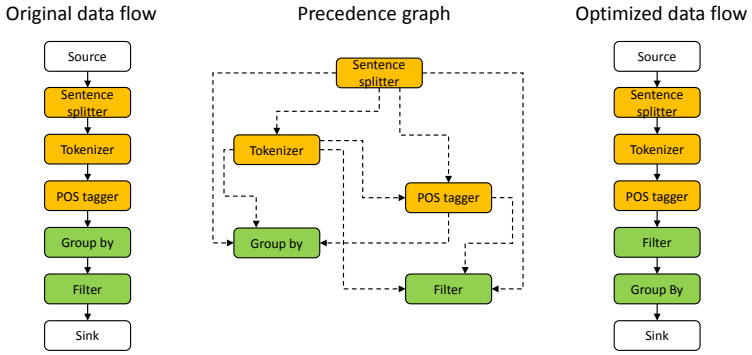


Figure 2.7: Precedence graph (left) and optimized data flow (right) for the example query of Figure 2.6. Precedence relationships of data source and data sink are omitted in the precedence graph to ease readability.

blocks. Reasoning along Presto relationships allows SOFA to automatically instantiate the templates with concrete operators and enables discovery of individual rewrite options for concrete operator combinations on the fly.

By extending the set of Sopremo operators, package developers and domain specialists can integrate application-specific functionality into Stratosphere as explained before. Likewise, by adding their new operators to Presto, they can enable cross-domain data flow optimization and extend the optimization potential of their operators in a pay-as-you-go manner.

Using Presto, Sofa performs three steps to enumerate alternative plans for a given data flow D: First, D is analyzed for precedence constraints between operators. This analysis yields a precedence graph P_D used in the plan enumeration phase to secondly enumerate, and thirdly to perform cost-based ranking of valid plan alternatives. Finally, the best plan is selected and compiled into a physical data flow program using the PACT programming model. As shown in the precedence graph in the middle of Figure 2.7, the data flow from Figure 2.6 contains only one degree of freedom, i.e., the `group by` and the `filter` operator are not in a precedence relationship with each other and can be reordered during optimization. A semantically equivalent and most likely more cost-effective plan is shown on the right side of the figure.

2.3.3 PACT programming model

For execution, each Sopremo operator of the optimized logical data flow is translated into one or more PACT operators. Before this step, each complex Sopremo operator is recursively decomposed into its components until only elementary operators remain. Elementary operators can be translated directly into second-order functions such as `map` and `reduce`, which are provided in the PACT programming model [Battré et al., 2010]. Alike Map/Reduce [Dean and Ghemawat, 2004], the PACT programming model grounds on second-order functions, each providing certain guarantees on what subsets of the input data will be processed together by the user-defined first-order function. PACT programs may consist of different parallelization primitives and the associated user-defined function. As shown in Figure 2.8, the PACT programming model consists

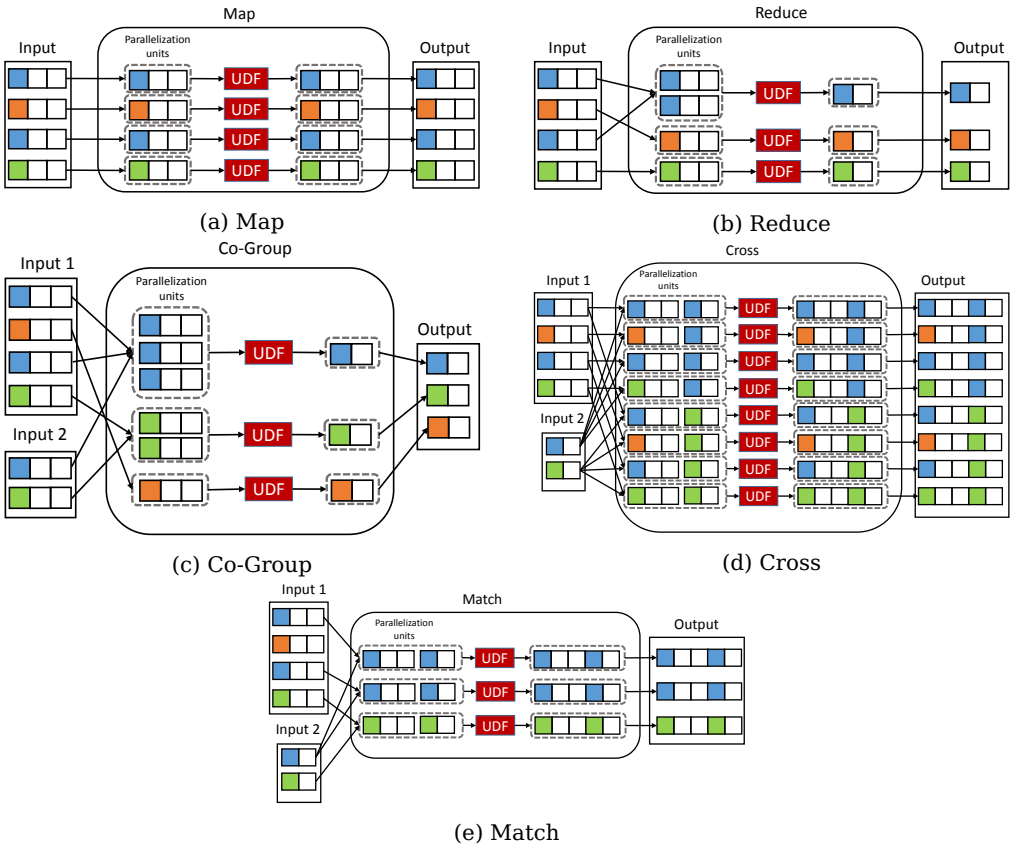


Figure 2.8: PACT parallelization primitives available in the Stratosphere system. Parallelization units of individual PACTs are indicated by dotted boxes.

of five second-order functions. These are the single-input functions `map` and `reduce` and three additional second-order functions, i.e., `match`, `cross`, and `cogroup`, to support an efficient implementation of multi-input operators:

- **`Map`** assigns each input record to an individual partition and independently processes each record with the user-defined first-order function.
- **`Reduce`** partitions the input based on a predefined key, which possibly consists of multiple attributes. All records that exhibit the same key value are handed together in one call to the same instance of the user-defined first-order function.
- **`Co-Group`** is a multi-input second-order function similar to Reduce. It partitions the records of all inputs based on a predefined key, such that all records with the same key form one partition, which is subsequently processed by the user-defined function.
- **`Cross`** computes a Cartesian product over its inputs and the user-defined first-order function is applied independently to each element of the Cartesian product.
- **`Match`** adheres to an inner equi-join semantics. It maps each record of the inputs to a single partition, such that records from different inputs, which have the same key, are processed together.

Each parallel data flow program is handed to the physical optimizer, which decides cost-based upon data exchange strategies and determines physical operator execution strategies in cases where multiple operator implementations are available (e.g., different join strategies).

2.3.4 Nephele execution engine

An optimized PACT program is compiled into a parallel execution plan, which is deployed on the given hardware by means of Nephele, a system for scheduling, executing, and monitoring DAG structured execution graphs on distributed systems [Warneke and Kao, 2009]. Similar to Sopremo and PACT, data flow programs for Nephele are DAGs where nodes represent individual tasks and edges model the data flow between tasks. However, Nephele data flow programs are equipped with a customized execution strategy, i.e., a suggested degree of parallelism for each task and data partitioning instructions, which adapts to the given compute environment and the data to be analyzed. Nephele is also responsible for executing data flow programs, in particular, it allocates necessary hardware resources for executing the data flow program, schedules individual tasks among the resources, monitors execution, and enables task recovery in the event of failure.

2.4 Summary

In this chapter, we presented fundamental terminology necessary for the remainder of this thesis followed by a summary of typical IE tasks and a discussion of existing

approaches and systems for IE at large scale. We also introduced the parallel data analytics system Stratosphere, its layered architecture, and the query and data flow compilation process in this system focussing on the Meteor query language and the algebraic layer Sopremo. In the next chapter, we introduce concrete operators for information extraction and web analytics designed and developed for Stratosphere together with their Meteor syntax and operator properties relevant for logical data flow optimization.

3 Operators for declarative text analytics

This chapter introduces logical operators for information extraction (IE) and web analytics (WA) designed and developed for the Stratosphere system. We discuss important aspects for designing non-relational operators for parallel data analytics systems and we present elementary and complex operators for IE and WA together with a description of their semantics, Meteor statements, configuration parameters, and concrete examples of usage.

3.1 Considerations for operator design

Before defining and implementing concrete operators for text analytics, we discuss different aspects that need to be taken into account for the operator design.

Declarative operator access through Meteor

The ultimate goal of designing IE and WA operators is to enable end-users to express and execute complex, non-relational data flows for domain-specific text analytics in a declarative, user-friendly way. Therefore, for each operator presented in this chapter, a declarative language primitive is available in Meteor. Each operator has a fixed interface and semantics and can be called with coherent, domain-independent expressions in the Meteor language.

Extensibility and adaptability to different IE goals and application domains

The need for operator adaption varies between the IE tasks to be performed, the concrete application domain, and the desired result quality in terms of precision and recall. For example, sentence splitting is typically performed domain-independent, whereas more advanced tasks of linguistic analysis (part-of-speech tagging, sentence parsing) and entity or relationship extraction are often implemented in a domain-specific way. Domain-adaptability of operator instantiations encompasses the provision of appropriate and configurable extraction models, rules, and dictionaries for the application domain at hand. Furthermore, the operators need to be designed in an extensible way to simplify the integration of new algorithms and domains. Therefore, all operators are instantiated to the concrete text analytics tasks and application domains at hand through concrete operator instantiations, which are configured with a set of formal operator properties available in Meteor.

Operator instantiations may call application- and domain-specific implementations, for example, to support different extraction goals (e.g., high recall, high throughput,

high precision), or to enable IE for concrete application domains. For all operators introduced in this chapter, we provide both general-purpose instantiations and highly specific instantiations for biomedical IE to showcase how operators can be tailored to concrete application domains.

Platform independency and scalability

Another important criterion for operator implementations is platform independency. Recall that the concrete way of data flow execution and distribution to a possibly heterogeneous compute environment is determined by Nephele, the scheduling and execution engine of Stratosphere, and thus, operator development for Sopremo needs to be agnostic of the underlying infrastructure. Since Stratosphere is a purely Java-based system, we decided to include only Java-based implementations, which can be made available to end-users in a user-friendly way through operator libraries. We decided to avoid including concrete operator implementations which are based on remote method invocations due to severe reliability issues of such calls in distributed systems (e.g., latency, network failure, setup and configuration of foreign code in heterogeneous systems) [Waldo et al., 1994]. All IE and WA operators available for Stratosphere are best-of-breed in their field, i.e., they are not specifically developed for the system, but adapted from existing IE and WA systems for parallel and distributed execution.

Operator modularization, interoperability, and self-containment to enable complex analytics

To enable complex analytics of unstructured data in Stratosphere, interoperability and modularization of operators is utterly important. All operators need to have a clear and well-defined semantics and need to be implemented in a self-contained and modular way. Operator modularization and design conceptually follows the set of IE tasks described in Chapter 2.2, i.e., for each IE task (e.g., sentence splitting, part-of-speech tagging, NER), a corresponding logical operator is available in Stratosphere. Elementary operators can be composed into complex operators providing access to larger IE tasks in a unified manner with a clearly defined semantics. Complex operators are partial data flows, which consume and produce annotated text collections. Introducing such operators has different advantages, namely (1) complex IE data flows become more concise, (2) components can be developed and debugged independently, which significantly reduces development effort, and (3) complex operators ensure that preconditions of contained elementary operators are met before execution.

Furthermore, interoperability between operators is ensured through the definition of a common data model. All operators for IE and WA build upon a common, semi-structured data model based on JSON (cf. Listing 2.1, Chapter 2.1). The input and output of each operator is a JSON record, which consists of text, annotations to the text, and possibly additional attributes required to support the concrete text analytics task (e.g. ID, publication date, authors, etc.). Unstructured text, which shall be analyzed, is stored in the string-typed JSON attribute *"text"*. Operators apply their methods to all records of input and use, add, or modify existing annotations, which are stored in the potentially nested and JSON attribute *"annotations"* of type object.

Listing 3.1: Elementary text segmentation operators.

```
using ie;
...
$article = annotate sentences $article;
$article = annotate tokens $article type 'sentence';
$article = annotate tokens $article type 'document';
$article = annotate ngrams $article type 'character' size 3;
$article = annotate ngrams $article type 'token' size 5;
...
```

3.2 Operators for information extraction

In this section, we introduce operators for all fundamental IE tasks described in Chapter 2.2. In the following, we give an overview of each logical operator, its instantiations, and usage examples. A complete list of all available elementary and complex IE operator instantiations is shown in Tables 2 and 4 of Appendix 1.

All IE operators can be categorized into an operator taxonomy as shown in Figure 3.1. Operators are categorized into classes of elementary and complex operators. The most abstract elementary operator for IE is in the `annott` operator for text annotation. Specializations can be distinguished between those performing text segment annotations, linguistic annotations, or semantic annotation of entities and relationships between entities. Each of these classes consists of multiple operators, for example, operators for annotating sentence boundaries (first class) part-of-speech tagging (second class), or for recognizing persons or companies and for detecting n-ary relationships between such entities (third class) are available. Specializations of `annott` write to designated attributes in the output record; for instance, all `annott-ent` operators for entity annotation write to a array-typed attribute *"entities"*, which is part of the JSON attribute *"annotations"*. Some specializations of `annott` are in precedence relations with other `annott` variants. For example, annotating relations between entities requires that entity annotations are already present in the input records.

Internally, each operator is implemented using a single Map function, and each operator consumes individual JSON records and produces for each processed record exactly one output record. All `annott` operators have an append-only semantics, i.e., they only add new annotations to the processed records but never delete or modify any existing annotations. For each class of IE tasks, we implemented operator instantiations based on the open-source IE and NLP framework OpenNLP, which provides a large collection of diverse IE algorithms for different applications. We also implemented a set of highly specialized operator instantiations to enable complex, domain-specific IE for applications in the biomedical domain.

3.2.1 Text segmentation

The set of text segmentation operators for Stratosphere contains seven operators, four of which are elementary and three of which are complex operators. Elementary operators comprise annotation operators, which identify different types of text segments in the input, and one auxiliary operator for splitting the input into segments. Complex op-

Figure 3.1: Taxonomy of elementary and complex IE operators. Concrete operator instantiations are not shown, but included in Tables 2 and 4 of Appendix 1.

Listing 3.2: Exemplary annotation of text segments for Listing 2.1.

```
1  { "id": "01",
2    "title": "1984",
3    "author": "George Orwell",
4    "text": "It was a bright cold day in April, and the clocks were striking thirteen.
5             Winston Smith, his chin nuzzled into his breast in an effort to escape the vile
6             wind, slipped quickly through the glass doors of Victory Mansions, though not
7             quickly enough to prevent a swirl of gritty dust from entering along with him.",
8    "annotation": {
9       "sentences": [{"start":0, "end":73, "sid":0}, {"start":75, "end":310, "sid":1},...],
10      "tokens":[{"sid":0,tok:[{"start":0, "end":1, "tid":0},
11                {"start":3, "end":5, "tid":1},...],...}],
12      "ngrams_char":[{"gid":0, "start":0, "end":2}, {"gid":1, "start":1, "end":3},...],
13      "ngrams_tok":[{"gid":0, "start":0, "end":19}, {"gid":1, "start":3, "end":23},}]
14   }
15 }
```

erators are composed of different combinations of elementary operators. In total, there are 11 operator instantiations available, which are described in more detail below.

Elementary operators

Listing 3.1 shows the Meteor statements for all elementary annotation operators for text segmentation. More concretely, sentence boundary detection (Line 3, operator `anntt-sent`), tokenization (Lines 4–5, operator `anntt-tok`), and annotation of n-grams (Lines 6–7, operator `anntt-ngram`) are shown.

Sentence boundary detection is based on the SentenceDetectorME class from OpenNLP, which uses a maximum entropy model to evaluate punctuation marks in a string to decide whether these marks constitute the end of a sentence. Sentence boundary detection does not require any annotation to be present in the input. Resulting annotations are shown in Listing 3.2, Line 9.

Detection of token boundaries is based on the TokenizerME class from OpenNLP, which uses a maximum entropy model trained for the English language to detect individual tokens in the input. Tokenization can either be performed sentence-wise (default) or document-wise by configuring the parameter `type`. If tokenization is performed sentence-wise, sentence boundaries need to be present in the input and the identified token annotations are grouped by sentence, as shown in Listing 3.2, Lines 10-11.

Detection of n-grams can either be performed based on tokens or on characters by configuring the `type` parameter of `anntt-ngrams`, as shown in the Meteor statements in Lines 6–7 of Listing 3.2. Gram size is also freely configurable through the parameter `size`. If none of the parameters are configured, the Meteor statement `annotate ngrams $text` annotates token n-grams of size 3 in the input $text. Character-based n-gram annotation does not require any annotation to be present in the input, however, n-gram annotation based on tokens requires information on token boundaries in the input (i.e., tokenization of the input needs to be performed beforehand). N-gram detection is performed using a self-developed algorithm, which linearly scans the input and recognizes all contained n-grams. Resulting n-gram annotations for character-based (Line 12) and token-based recognition (Line 13) are shown in Listing 3.2.

Listing 3.3: Text segmentation into sentences based on annotated sentence boundaries shown in Listing 3.2.

```
1 using ie;
2 ...
3 $article = split text on 'sentences';
4 ...
```

Next to `anntt` operators for identifying individual text segments, the set of IE operators for Sopremo also contains an operator `splt-txt` for splitting unstructured text into recognized text segments. By configuring the operator property `on`, one can specify whether the input is splitted into previously recognized sentences, tokens, or n-grams. The Meteor statement for splitting the JSON record of Listing 3.2 into sentences is shown in Listing 3.3 and exemplary output for this operation is shown in Listing 3.4. The `splt-txt` operator is implemented in a Map second-order function processing each input record individually. In contrast to operators annotating segment boundaries, this operator creates for each input record n output records, where n is the number of annotated text segments to be splitted. Thus, for the exemplary JSON record from Listing 3.2, two individual records are created. During split, substrings of the input text stored in the attribute *"text"* are determined based on the previously annotated sentence boundaries and new JSON records are created for each substring. During record creation, existing text annotations are filtered for the respective substring, index positions are adjusted, and all other attributes (e.g.,*"id", "title", "author"*) are copied without modification as shown in Listing 3.4.

Complex operators

Recall that complex operators consist of combinations of multiple elementary operators to provide shortcuts to complex IE tasks. For text segmentation, we developed three complex operators based on the elementary annotation operators discussed in the previous paragraph.

Complex operators for text segmentation are shown in Listing 3.5. The operator `splt-sent` performs sentence splitting, `splt-tok` splits text into tokens, and the operator `splt-ngrams` splits text into n-grams. Each of these operators first determines sentence, token, and n-gram boundaries, respectively, for each input record and splits the records afterwards according to the detected text segments. Analog to the elementary operators for n-gram detection, `splt-ngrams` provides two operator instantiations for n-gram detection on token and character level, and `splt-tok` provides two instantiations for tokenization on document and character level. Internally, the complex operators for text segmentation each consist of two operators, one annotation operator and one split operator configured according to the segment type. Overall, the semantics of complex text segmentation operators adheres to the semantics of `splt-txt`, i.e., for each input record n output records are created and the input text is splitted into n substrings, and text segment annotations are provided for each created JSON record.

Listing 3.4: Exemplary split of JSON record from Listing3.2 into sentence segments.

```
[{ "id": "01",
   "sid": 0
   "title": "1984",
   "author": "George Orwell",
   "text": "It was a bright cold day in April, and the clocks were striking thirteen.",
   "annotation": {
      "sentences": [{"start":0, "end":73, "sid":0}, {"start":75, "end":310, "sid":1},...],
      "tokens":[{"sid":0,tok:[{"start":0, "end":1, "tid":0}, {"start":3, "end":5, "tid
          ":1},...],...}],
      "ngrams_char":[{"gid":0, "start":0, "end":2}, {"gid":1, "start":1, "end":3},...],
      "ngrams_tok":[{"gid":0, "start":0, "end":19}, {"gid":1, "start":3, "end":23},...]
    }
 },
 { "id": "01",
   "sid": 1,
   "title": "1984",
   "author": "George Orwell",
   "text": "Winston Smith, his chin nuzzled into his breast in an effort to escape the vile
    wind, slipped quickly through the glass doors of Victory Mansions, though not quickly
    enough to prevent a swirl of gritty dust from entering along with him.",
   "annotation": {
      "sentences": [{"start":0, "end":235, "sid":1}],
      "tokens":[{"start":0, "end":6, "tid":0}, {"start":8, "end":13, "tid":1},...],
      "ngrams_char":[{"gid":0, "start":0, "end":2}, {"gid":1, "start":1, "end":3},...],
      "ngrams_tok":[{"gid":0, "start":0, "end":30}, {"gid":8, "start":34, "end":23},...]
    }
 }]
```

Listing 3.5: Meteor statements for complex text segmentation operators.

```
using ie;
...
$article = split sentences $text;
$tokens = split tokens $text;
$ngrams = split ngrams $text;
...
```

Listing 3.6: Elementary operators for linguistic analysis.

```
1 using ie;
2 ...
3 $article = annotate postags $article use algorithm 'treetagger';
4 $article = annotate stems $article;
5 $article = annotate stopwords $article type 'english';
6 $article = annotate parse $article;
7 ...
```

3.2.2 Linguistic analysis

The set of IE operators for linguistic analysis comprises eight operators, six of which are elementary and two of which are complex. Elementary operators determine linguistic structures, such as part-of-speech tags, word stems, stopwords, and parse trees in the input records and annotate those records with the retrieved linguistic information. Complex operators perform more advanced operations such as the combined identification and removal of stopwords. In total, the set of operators for linguistic text analysis comprises 13 operator instantiations described in more detail below.

Elementary operators

Listing 3.6 displays the Meteor statements for all elementary annotation operators, which analyze the input linguistically: part-of-speech tagging (`anntt-pos`, Line 3), stemming (`anntt-stem`, Line 4), stopword detection (`anntt-stop`, Line 5), and sentence parsing (`anntt-parse`, Line 6). All these Meteor statements are implemented in a single Map second-order function and consume and produce individual JSON records. All annotation operators have an append-only semantics.

Part-of-speech tagging can either be performed with the general purpose tool TreeTagger [Schmid, 1995], which can be configured for many different languages or with a specialized algorithm for tagging English text based on the OpenNLP tagger. Furthermore, we included Medpost [Smith et al., 2004], a part-of-speech tagger for the biomedical domain. Medpost and TreeTagger employ Markov models for tagging, while tagging with OpenNLP is performed using a maximum entropy model. The concrete tagging algorithm can be specified by configuring the operator property `use algorithm` with the strings *'opennlp'*, *'medpost'*, or *'treetagger'* as shown in Line 3 of Listing 3.6. If no algorithm is specified, the OpenNLP part-of-speech tagger is used as default. Exemplary part-of-speech annotations are shown in Lines 9–10 of Listing 3.7. Before part-of-speech tagging is performed, detection of sentence and token boundaries needs to be applied such that annotations of sentence and token boundaries are present in the input for `anntt-pos`.

The operator `anntt-stem` for determining root forms (stems) of words is based on the Porter stemmer algorithm [Porter, 1997] included in OpenNLP. This algorithm applies a set of word shortening rules, which are applied in a determined order until the stemmed word is reduced to a minimum number of syllables. Although the Porter stemmer algorithm is originally developed for English text, it can be extended in a straight forward way to other languages by providing different sets of stemming rules. The Meteor statement for `anntt-stem` is shown in Line 4 of Listing 3.6 and the produced output is shown

Listing 3.7: Exemplary linguistic annotation for the JSON record shown in Listing 3.2.

```
1  { "id": "01",
2    "title": "1984",
3    "author": "George Orwell",
4    "text": "It was a bright cold day in April, and the clocks were striking thirteen. ...",
5    "annotation": {
6       "sentences": [{"start":0, "end":73, "sid":0}, {"start":75, "end":310, "sid":1},...],
7       "tokens": [{"sid":0,tok:[{"start":0,"end":1,"tid":0}, {"start":3,"end":5,"tid":1},
8                  {"start":7,"end":7,"tid":2}, ...]}],
9       "postags": [{"sid":0,"tid":0,"tag":"PRP"}, {"sid":0,"tid":1,"tag":"VBD"},
10                  {"sid":0,"tid":2,"tag":"DT"},...],
11      "stems": [{"sid":0,"stem":"It","tid":0}, {"sid":0,"stem":"wa","tid":1},
12                {"sid":0, "stem":"a","tid":2}, ...],
13      "stopwords": [{"sid":0,"tid":0}, {"sid":0,"tid":1}, {"sid":0,"tid":2},
14                    {"sid":0,"tid":6},...],
15      "parse-tree": [{"sid":0, "start":0, "end":73, "tree":"(S (S (NP It) (VP was (NP a
16                      bright cold day) (PP in (NP April)))), and (S (NP the clocks)
17                      (VP were (VP striking (NP thirteen)))) .)", "type":"constituent"}, ...]
18   }
19 }
```

in Lines 11–12 of Listing 3.7. Similar to part-of-speech tagging, sentence splitting and tokenization has to be perfomed prior to stemming.

The `anntt-stop` operator for stopword detection is implemented with a self-developed algorithm, which matches all recognized tokens in the given input against a dictionary of stopwords. The concrete stopword list can be configured with the property *type* as shown in Line 5 of Listing 3.6 and currently, a list for the English language and a domain-specific list for biomedical documents[5] is available. For each detected stopword, the `anntt-stop` operator annotates sentence and token ID of the stopwords in the analyzed text (cf. Lines 13–14 of Listing 3.7). Tokenization has to be performed prior to stopword detection.

The `anntt-parse` operator for sentence parsing builds a constituency parse tree for each tokenized sentence of the input text. Internally, it employs the parsing algorithm and models for English texts provided by OpenNLP. The Meteor statement for this operator is shown in Line 6 of Listing 3.6 and the corresponding annotations produced are shown in Lines 15–17 of Listing 3.7.

Stratosphere contains two additional operators for transforming the input text based on annotated linguistic information: The operator `repl-stem` replaces individual tokens with their corresponding word stems. Note that the abstract operator `repl` can also be configured to replace identified entities with their normalized variants (see Section 3.2.3). In the same manner, the `rm-stop-anntt` operator removes recognized stopwords from the input based on the available stopword annotations. Since both `repl-stem` and `rm-stop-anntt` modify the input text by removing or replacing tokens, all text segment annotations are invalidated and thus removed. Analog to the annotation operators described previously, both `repl` and `rm-stop-anntt` operator are each implemented in a single Map second-order function.

[5] The set of stopwords for biomedical documents was obtained from `http://www.ncbi.nlm.nih.gov/books/NBK3827/table/pubmedhelp.T.stopwords/`, last accessed: 2016-11-31.

Listing 3.8: Complex operator for stopword removal.

```
using ie;
...
$article = remove stopwords $article type 'english';
...
```

Listing 3.9: Stopword removal with the complex `remove stopwords` operator for the JSON record shown in Listing 3.2.

```
{ "id": "01",
  "title": "1984",
  "author": "George Orwell",
  "text": "bright cold day April, clocks striking thirteen. ...",
  "annotation": {
     "stopwords": [{"sid":0,"tid":0}, {"sid":0,"tid":1}, {"sid":0,"tid":2},
                   {"sid":0,"tid":6},...],
  }
}
```

Complex operators

Two complex operators for analyzing texts and transforming them based on linguistic information are available, i.e., `rm-stop` for retrieving and removing stopwords as shown in Listing 3.8 and `stem` for text stemming as shown in Listing 3.10.

The `rm-stop` operator consists of two consecutive elementary operators, i.e., `anntt-stop` followed by `rm-stop-anntt`, and it can be configured for usage with different text types in the same way as the `anntt-stop` operator. If the example JSON record shown in Listing 3.7 is processed by `rm-stop` (cf. Listing 3.9), stopwords are first determined, the attribute *text* is modified and stopwords are removed. Finally, since all existing linguistic annotations are invalidated, these annotations are deleted and the JSON record shown in Listing 3.9 is produced.

The `stem` operator has a similar structure as `rm-stop`, i.e., it consists of the two elementary operators `anntt-stem` and `repl-stem`. If the example JSON record shown in Listing 3.7 is processed by `stem`, word stems are first determined for each token and each token in the attribute *text* is replaced with its corresponding word stem. Analog to `rm-stop`, all linguistic annotations are now invalidated by the replacement operation, the corresponding annotations are deleted, and the JSON record shown in Listing 3.11 is returned.

3.2.3 Named entity and relationship recognition

The set of IE operators for named entity and relationship recognition comprises five operators, three of which are elementary and two of which are complex. Most elementary operators retrieve mentions of entities and relationships in the input records and annotate those records with the respective start and end positions and a description of the specific entity or relationship, respectively. Complex operators not only retrieve entities and relationships but actually extract the found mentions, i.e., those operators perform a complex transformation of the input JSON records into a different semi-structured

Listing 3.10: Complex operator for stemming.

```
1 using ie;
2 ...
3 $article = stem $article;
4 ...
```

Listing 3.11: Stemming with the complex `stem` operator for the JSON record shown in Listing 3.2.

```
1 { "id": "01",
2   "title": "1984",
3   "author": "George Orwell",
4   "text": "It wa a bright cold dai in April and the clock were strike thirteen",
5   "annotation": {
6       "stems": [{"sid":0,"stem":"It","tid":0}, {"sid":0,"stem":"wa","tid":1},
7                 {"sid":0, "stem":"a","tid":2}, ...],
8   }
9 }
```

format. In total, the set of operators for named entity and relationship recognition comprises 44 operator instantiations, which will be described below.

Elementary operators

Listing 3.12 displays exemplary Meteor statements for the elementary entity and relationship annotation operators `anntt-ent` and `anntt-rel`. Analog to other elementary annotation operators described in the previous sections, each entity and relationship annotation operator is implemented in a single Map second-order function, consumes and produces individual JSON records, and has an append-only semantics. Entity annotation can be carried out with different algorithms by specifying the property `use algorithm`. Specifically, algorithms exist for

- automaton-based matching of dictionaries (`'linnaeus'`, cf. Line 3 of Listing 3.12),
- exact matching of regular expressions (`'regex'`, cf. Line 4 of Listing 3.12),
- matching of plain text strings (`'exact'`), which are either provided from file or directly in the Meteor statement,
- machine-learning based entity recognition using the NameFinder class of OpenNLP (`'opennlp'`, cf. Lines 7–8 of Listing 3.12), and
- different domain-specific algorithms for annotating biomedical entities (e.g., gene name recognition with `'banner'`, drug recognotion with `'wbi-drug'`, disease recognition with `'tr-disease'`).

Some entity annotation algorithms are specially designed for a certain entity type (e.g., `'banner'` [Leaman and Gonzalez, 2008] and `'chemspot'` [Rocktäschel et al., 2012] annotate genes and chemicals, respectively). Other algorithms are able to detect different types of entities (e.g., `'linnaeus'` or `'regex'`). In these cases, the concrete entity

Listing 3.12: Elementary operators for entity and relationship detection.

```
1  using ie;
2  ...
3  $article = annotate entities $article use algorithm 'linnaeus' type 'disease';
4  $article = annotate entities $article use algorithm 'regex' with '19|20[0-9][0-9]'
5     type 'date';
6  $article = annotate entities $article use algorithm 'opennlp' type 'person';
7  $article = annotate relations $article use algorithm 'co-occurrences'
8     type ['person', 'date'];
9  $article = annotate relations $article use algorithm 'co-occurrences'
10    type ['person', 'disease'];
11 ...
```

type must be specified to ensure that the appropriate dictionaries or models are loaded by specifying the operator property `type`. Models and algorithms for seven different general-purpose entity types (`'person'`, `'date'`, `'location'`,`'money'`, `'organization'`, `'percentage'`, and `'time'`) and eight different biomedical entity types (`'cell'`, `'compound'`, `'disease'`, `'drug'`, `'enzyme'`, `'gene'`, `'species'`, and `'tissue'`) are available. Depending on the entity type and algorithm used for annotation, the instantiations of `anntt-ent` have different dependencies on other operators. For example, automaton-based annotation of gene names requires sentence and token boundary annotations in the input, whereas the domain-specific algorithm Banner [Leaman and Gonzalez, 2008] only requires sentence boundary annotations. For a complete list of such prerequisites, see Table 5 in Appendix 1, which lists operator properties and dependencies for all elementary IE operators. An example text mentioning different types of entities and resulting annotations for the entity types `'disease'`, `'date'`, and `'person'` are shown in Lines 4–11 of Listing 3.13.

Relationship detection is carried out with the operator `anntt-rel` domain-independently with a co-occurrence based algorithm, detecting n-ary relationships between entities occurring in the same context (e.g., sentence, document, paragraph). Currently, `anntt-rel` is available for sentence-based relationship detection, but can easily be extended for different contexts by specifying and implementing the operator property `in context` with different types. Exemplary Meteor statements of `anntt-rel` for detecting relationships between persons and diseases and between persons and dates are shown in Lines 7–10 of Listing 3.12. Concrete entity types, for which relationships shall be detected, are specified by configuring the `type` property with an array of entity types. Relationship detection requires that entity annotations for all entity types in the concrete relation are present in the input and the resulting annotations for both relationship types of Listing 3.12 are shown in Lines 12–18 of Listing 3.13.

Apart from entity and relationship annotation, an operator `mrg` for merging existing annotations is available. This operator merges two JSON records a, b from two input sets A, B based on a user-defined merge condition (e.g., document ID). We found `mrg` to be helpful in data flows, where multiple entity or relationship detection operators are executed in an inter-operator-parallel way. In contrast to a join operator, `mrg` has an append semantics for existing annotations, i.e., if the JSON records for a and b both have entity annotations, the resulting record c contains a single attribute with all annotations from a and b.

Listing 3.13: Exemplary entity and relationship anntoation.

```
{ "id": "0",
  "text": "The film director Curtis Hanson was born in 1945. Hanson was reported to have
      dementia and died in 2016.",
  "annotation": {
      "entities":[{"sid":0,"eid":"Curtis Hanson","start":18,"end":30,"text":"Curtis Hanson",
                  "algorithm"="opennlp", "type"="person"}, {"sid":1,"eid":"Curtis Hanson",
                  "start":0,"end":5,"text":"Hanson","algorithm"="opennlp","type"="person"},
                  {"sid":1,"eid":"MeSH:C10.228.140.380","start":28,"end":35,
                  "text":"dementia","algorithm"="linnaeus", "type"="disease"},
                  {"sid":0,"eid":"1945","start":44,"end":47,"text":"1945",
                  "algorithm"="regex", "type"="date"}, {"sid":1,"eid":"2016","start":50,
                  "end":53,"text":"2016", "algorithm"="regex", "type"="date"}],
      "relations":[{"sid":0,"eAid":"Curtis Hanson","eAstart":18,"eAend":30,"eBid":"1945",
                   "eBstart":44,"eBend":47, "type"="[person, date]", "algorithm"="co-occ"},
                   {"sid":1,"eAid":"Curtis Hanson","eAstart":0,"eAend":5,"eBid":"2016",
                   "eBstart":50,"eBend":53, "type"="[person, date]", "algorithm"="co-occ"},
                   {"sid":1,"eAid":"Curtis Hanson","eAstart":0,"eAend":5,
                   "eBid":"MeSH:C10.228.140.380","eBstart":28,"eBend":35,
                   "type"="[person, disease]", "algorithm"="co-occ"}],
      "tokens":[...],
      "sentences":[...]}
}
```

Listing 3.14: Elementary operators for extracting entity and relationship annotations.

```
1 using ie;
2 ...
3 $entities = emit entities $article;
4 $relations = emit relations $article;
5 ...
```

Using the operator `emit` (cf. Listing 3.14), existing entity or relationship annotations are extracted from the incoming JSON records and transformed into the format shown in Listing 3.15. In this example, entity annotations from Listing 3.13 are extracted using the Meteor statement shown in Line 3 of Listing 3.14.

Complex operators

Two complex operators, `extr-ent` and `extr-rel`, are available for named entity and relationship extraction as shown in Listing 3.16. Internally, both operators consist of an `anntt` operator followed by an `emit` operator to first annotate respective entities or relations and subsequently transform the input record into the format shown in Listing 3.15. The semantics of the complex `extract entities` and `extract relations` opertors adhere to the semantics of `emit`, i.e., the incoming record is transformed into a different output format if entities or relationships were detected. If no entity or relationship was detected, the complex operator returns an empty result. `Extr-ent` and `extr-rel` have the same prerequisites as the contained annotation operators, for example, gene name recognition with the algorithm "Banner" requires sentence annotations to be present.

Listing 3.15: Exemplary output of `emit` operator.

```
[{"id":0, "sid":0,"eid":"Curtis Hanson","start":18,"end":30,"text":"Curtis Hanson",
    "algorithm"="opennlp", "type"="person"},
 {"id":0, "sid":1,"eid":"Curtis Hanson","start":0,"end":5,"text":"Hanson",
    "algorithm"="opennlp","type"="person"},
 {"id":0, "sid":1,"eid":"MeSH:C10.228.140.380","start":28,"end":35, "text":"dementia",
    "algorithm"="linnaeus", "type"="disease"},
 {"id":0, "sid":0,"eid":"1945","start":44,"end":47,"text":"1945",
    "algorithm"="regex", "type"="date"},
 {"id":0, "sid":1,"eid":"2016","start":50,"end":53,"text":"2016",
    "algorithm"="regex", "type"="date"}]
```

Listing 3.16: Examples of complex operators for entity and relationship detection.

```
using ie;
...
$entities = extract entities $article use algorithm 'linnaeus' type 'disease';
$entities = extract relations $article use algorithm 'co-occurrences'
   type ['person', 'disease'];
...
```

3.3 Operators for web analytics

In this section, we introduce operators for processing and analyzing HTML pages with Stratosphere[6]. The set of web analytics (WA) operators comprises operators for preprocessing of HTML documents to enable downstream IE on such documents, i.e., operators for HTML markup detection, repair, and removal, but also operators for detecting structured information in HTML pages (link URLs, tables, lists). All operators described below process individual JSON records as shown in the exemplary record contained in Listing 3.17. Analog to text processing with IE operators, we require the unstructured HTML documents to be stored in the attribute "*text*".

3.3.1 Text preprocessing

The set of WA operators for text preprocessing consists of three elementary and one complex operator: The elementary operators perform HTML boilerplate detection (`dtct-bp`), markup repair (`rpr-mrk`), and markup removal (`repl-mrk`), while the complex operator `rm-mrk` performs markup repair, boilerplate detection, and removal in a single step. In total, the set of operators for preprocessing HTML pages comprises 14 operator instantiations, which are described below in more detail:

Listing 3.18 lists all Meteor statements for preprocessing of HTML web pages. The third line of the script displays the `rpr-mrk` operator for repairing erroneous markup tags in the input. This step is important for many real-life text analytics tasks, since many websites contain markup errors, which pose severe challenges to boilerplate detection operators [Ofuonye et al., 2010]. Internally, `rpr-mrk` transforms the *text* attribute of the input record by either reordering, deleting redundant HTML tags or in-

[6]The set of web analytics operators was implemented by Jörg Meier and Anja Kunkel under close supervision based on the requirements and specifications provided by Astrid Rheinländer.

Listing 3.17: Exemplary JSON record for HTML pages.

```
{ "id": "http://www.test.com/test.html",
  "text": "<!DOCTYPE html><html><body>This is a simple HTML page with an absolute URL
           <a href=\"http://www.w3schools.com\">W3Schools</a> and a relative URL: <a
           href=\"tags/tag_a.asp\">The a tag</a>. The page contains an example of a
           table: <br>
           <table border="1"><thead><tr><th>Month</th>
           <th colspan="2">Savings</th></tr></thead><tr><td>January</td><td>100</td>
           <td rowspan="2">$</td></tr><tr><td>March</td><td>3400</td></tr></table>
           <br><br>
           Examples of lists are also available:
           <ul><li>Coffee</li>
               <li>Tea</li>
               <li>Milk</li></ul>
           <ol start=\"50\">
               <li>Soda</li>
               <li>Water</li>
               <li>Oil</li></ol>
           </body></html>"
  }
}
```

serting missing tags to produce valid XML code using the HTMLCleaner algorithm[7]. The operator is implemented in a *Map* second-order function and produces a single output record for each processed input record.

The boilerplate detection operator `dtct-bp` aims at detecting the net content of web pages and the corresponding Meteor statement is shown in Line 4 of Listing 3.18. We implemented six different operator instantiations for `dtct-bp`, i.e., three algorithms provided by the Boilerpipe[8] library [Kohlschütter et al., 2010], two algorithms provided by Readability[9], and one algorithm called cuter developed by Krause [2012][10]. Each algorithm can be called in Meteor by configuring the `dtct-bp` operator with the `use algorithm` property. The algorithms `'blp'`,`'blp_largest'`, and `'blp_news'` are taken from the Boilerpipe library. `Blp` is the default algorithm for `dtct-bp` due to its robustness and performance on arbitrary HTML pages. `Blp_largest` detects the largest content block contained in a web page, while blp_news is specifically designed for processing news pages. The algorithms `'snack'` and `'rdb'` both identify non-content elements reliably, but might also classify parts of the actual content as non-content elements. In contrast, `'cuter'` detects all content elements reliably, but might also classify many non-content elements as actual content. After content has been detected, the `dtct-bp` operator creates a new attribute *detected_content* as shown in Listing 3.19.

The third operator for text preprocessing is `repl-bp`, which replaces the "*text*" attribute of the input record with the content detected by `dtct-bp`. At the same time, the attribute *detected_content* created by `dtct-bp` is deleted. The corresponding Meteor

[7] HTMLCleaner `http://htmlcleaner.sourceforge.net/`, last accessed: 2016-09-26.

[8] Boilerplate Removal and Fulltext Extraction from HTML pages, `http://github.com/kohlschutter/boilerpipe`, last accessed 2016-09-26.

[9] Readability web parser API, `http://www.readability.com`, last accessed: 2016-09-26.

[10] Before operators were implemented for Stratosphere, Jörg Meier conducted an pre-study under close supervision by Astrid Rheinländer. This study compared the extraction quality and runtime properties of all implemented boilerplate detection algorithms and is available upon request. Exemplary results regarding extraction quality are contained in Appendix 4 of this thesis.

statement for this operator is shown in Line 5 of Listing 3.18 and exemplary output is shown in Listing 3.20.

Listing 3.18: Elementary web analytics operators.

```
1 using wa;
2 ...
3 $content = repair markup $htmlcode;
4 $content = detect content $content use algorithm 'blp';
5 $content = replace content $content;
6 ...
```

Listing 3.19: Output of `dtct-bp` operator for the HTML page shown in Listing 3.17.

```
1 { "id": "http://www.test.com/test.html",
2   "text": "<!DOCTYPE html><html><body>This is a simple HTML page with an ..."
3    "detected_content": [{"algorithm":"blp","content":"This is a simple HTML page with
4            an absolute URL..."}]
5   }
6 }
```

Listing 3.20: Output of `repl-bp` operator for the HTML page shown in Listing 3.17.

```
1 { "id": "http://www.test.com/test.html",
2   "text": "This is a simple HTML page with an absolute URL..."
3 }
```

Listing 3.21: HTML document preprocessing with the complex `rm-bp` operator.

```
1 using wa;
2 ...
3 $content = remove markup $htmlcode use algorithm 'snack';
4 ...
```

One complex operator `rm-bp` for preprocessing of HTML documents is available, which performs HTML markup repair, boilerplate detection, and boilerplate removal in a single step[11]. The corresponding Meteor statement is shown in Listing 3.21. Internally, this operator consists of three operators, i.e., `rpr-mrk` is followed by `dtct-bp` and `repl-bp`. Analog to `dtct-bp`, the complex operator `rm-bp` can be configured for using one of the six boilerplate detection methods available for `dtct-bp`.

3.3.2 Structure detection

For structure detection, one elementary operator `dtct-struct` with three concrete instantiations is available: `dtct-struct-link` for detecting URL links, `dtct-struct-tbl` for detecting tables, and `dtct-struct-lst` for detecting lists in the input records. Note that these operators operate on the original HTML input, i.e., in complex data flows, boilerplate detection needs to be performed after link, table, and list detection.

The operator `dtct-struct-link` retrieves and extracts all links inside the *<body>* scope of an HTML page based on the HTML parser provided by Jsoup[12]. The Meteor

[11] Note that markup repair is included in this algorithm for practical reasons, since most web pages available online contain errors, which may cause severe issues in the downstream boilerplate detection phase [Ofuonye et al., 2010].

[12] jsoup: Java HTML Parser, `http://jsoup.org/`, last accessed: 2016-09-26.

statement for `dtct-struct-link` is shown in Line 3 of Listing 3.22. As shown in Lines 3–4 of Listing 3.23, the operator extracts all absolute and relative links. If possible, relative links are completed with the base URI of the processed input record provided that a source URL is available for the respective record.

The operator instantiation `dtct-struct-table` retrieves and extracts all tables inside the *<body>* scope of an HTML page using Jsoup. The Meteor statement for `dtct-struct-link` is shown in Line 4 of Listing 3.22 and exemplary output is shown in Lines 5–12 of Listing 3.23. By creating additional attributes, which describe the table region, cell type, and column/row IDs, the structure of the extracted table is preserved with respect to its layout in the original HTML document.

For extracting ordered and unordered lists, the operator instantiation `dtctstruct-lst` analyzes the *<body>* scope of an HTML page using Jsoup. Every list is extracted into the output format shown in Lines 13–15 of Listing 3.23. For ordered lists, the item order is preserved by adding a position attribute for each list item.

Listing 3.22: Structure detection operators for HTML documents.

```
1 using wa;
2 ...
3 $content = extract html $htmlcode use type 'url';
4 $content = extract html $htmlcode use type 'table';
5 $content = extract html $htmlcode use type 'list';
6 ...
```

Listing 3.23: Output of `dtct-bp` operator for the HTML page shown in Listing 3.17.

```
1  { "id": "http://www.test.com/test.html",
2    "text": "<!DOCTYPE html><html><body>This is a simple HTML page with an ...",
3    "links":[{"url":"http://www.w3schools.com", "desc": "W3Schools"},
4             {"url":"http://www.w3schools.com/tags/tag_a.asp", "desc": "The a tag"}],
5    "tables":[{"table_id" : 1, "content" :[
6                {"col":0,"row":0,"text":"Month","reg":"thead","ctype":"th"},
7                {"col":1,"row":0,"text":"Savings","reg":"thead","ctype":"th","colspan":"2"},
8                {"col":0,"row":1,"text":"January","reg":"tbody","ctype":"td"},
9                {"col":1,"row":1,"text":"100","reg":"tbody","ctype":"td"} ,
10               {"col":3,"row":1,"text":"$","reg":"tbody","ctype":"td","rowspan":"2"},
11               {"col":0,"row":2,"text":"March","reg":"tbody","ctype":"td"} ,
12               {"col":1,"row":2,"text":"3400","reg":"tbody","ctype":"td"}]}],
13   "lists":[{"list_id":0, "items":["Coffee","Tea","Milk"] },
14            {"list_id":1, "items":[{"position":0, "content":"Soda"},
15               {"position":1, "content":"Water"}]}]
16 }
```

3.4 Functional and runtime operator properties

All operators introduced in the previous sections have different characteristics in terms of semantics, prerequisites, algebraic properties, the concrete algorithm implemented for creating an operator instantiation, the way how the schemata of consumed and produced JSON records are treated, and the ratio between the number of input and output properties. As we will explain in Chapters 4 and 5, these properties play an important role during optimization of data flows with non-relational operators. Table 3.1

ID	Operator	Algorithm	Type	Prerequisites
E1	`anntt-sent`	OpenNLP	–	–
E2a	`anntt-tok`	OpenNLP	sentence	E1
E2b	`anntt-tok`	OpenNLP	document	–
E3a	`anntt-ngram`	own	token	E2
E3b	`anntt-ngram`	own	character	–
E4a	`splt-txt`	own	sentence	E1
E4b	`splt-txt`	own	token	E2*
E4c	`splt-txt`	own	ngram	E3*

Table 3.1: Elementary Sopremo operator instantiations for text segmentation with associated algorithms, types, and prerequisites.

ID	PACT(s)	I/O ratio	Record size	Schema handling	Processing type	Idempotent	Commutative with
E1	Map	$I = O$	$I \leq O$	extension	RAAT		E1
E2*	Map	$I = O$	$I \leq O$	extension	RAAT		E2*,E3*
E3a	Map	$I = O$	$I \leq O$	extension	RAAT		E3*
E3b	Map	$I = O$	$I \leq O$	extension	RAAT		E2*,E3*
E4a	Map	$I \leq O$	$I \geq O$	modification	RAAT	✓	E11*, E12
E4b,c	Map	$I \leq O$	$I \geq O$	modification	RAAT		

Table 3.2: Properties of elementary text segmentation operators.

displays algorithms, types, and prerequisites and Table 3.2 displays functional properties of elementary IE operators for text segmentation. In the tables, each operator instantiation is identified by an ID and corresponding algorithms, operational types, and functional properties are listed. For example, the instantiations for the logical operator `anntt-ngram` are identified by the IDs `E3a` and and `E3b`, respectively. Both instantiations differ by their operational type and prerequisites, i.e., `E3a` annotates token n-grams and requires token annotations, whereas `E3b` annotates character n-grams and does not have any prerequisites. As shown in Table 3.2, both instantiations are record-at-a-time operators implemented in a Map function, which produce for each input record exactly one output record. Furthermore, both instantiations may extend the schema of incoming records and max extend the size of the input record by producing new annotations. Both instantiations are also commutative with other operator instantiations, i.e., `E3a` is commutative with any instantiation of `E3` (indicated by the '*' symbol) and `E3b` is commutative with any instantiation of `E2` and `E3`. For all remaining elementary and complex IE and WA operators, a complete overview of these characteristics can be found in tabular form in Appendix 1.

In addition to these functional properties, the operator instantiations differ heavily in terms of execution and startup times. To assess the IE and WA operators available for Stratosphere in terms of runtime characteristics, we randomly sampled 10,000 documents from Medline[13], 10,000 documents from a set of plain-text documents of Wikipedia available in English[14], and 1,000 HTML documents from the data sets News-600, GoogleNews, and CleanEval used for evaluating the quality of boilerplate detection

[13] U.S. National Library of Medicine, `http://www.ncbi.nlm.nih.gov/pubmed`, last accessed: 2016-12-09.
[14] The Wikimedia foundation, `http://dumps.wikimedia.org`, last accessed: 2016-12-09.

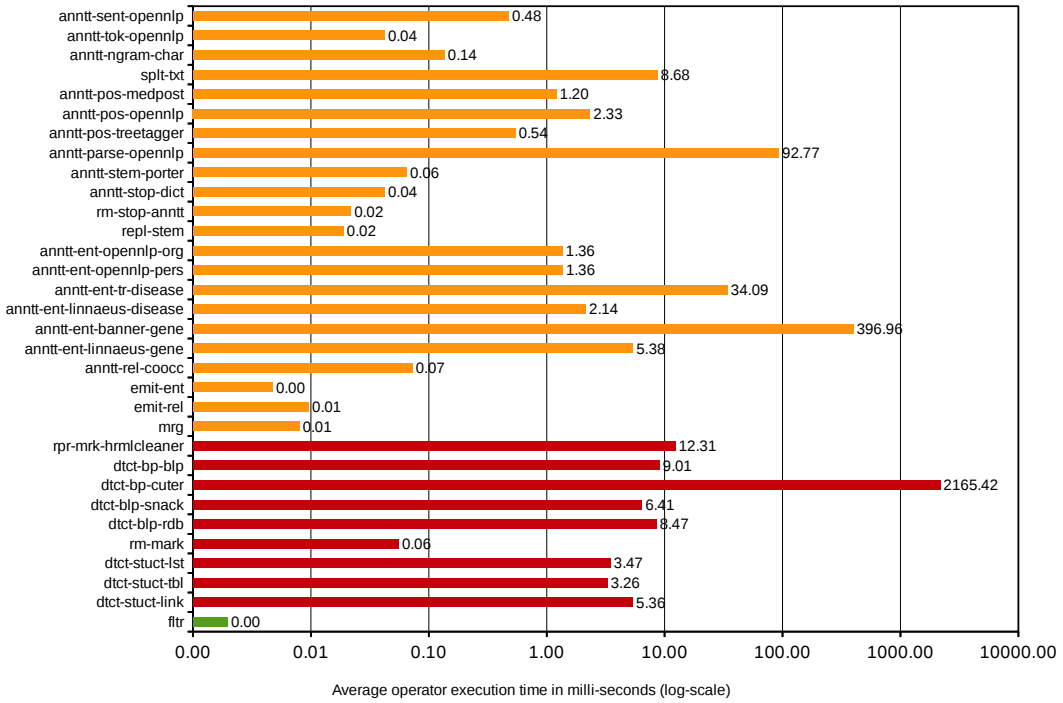


Figure 3.2: Average execution time of IE and WA operators (log-scale). Yellow bars show concrete IE operator instantiations, red bars show concrete WA operator instantiations. For comparison, the average execution time of the Meteor operator `fltr` for filtering records (green bar) is also shown. Note that measurements are rounded to two decimal places.

operators from the WA package (cf. Appendix 4). All experiments were conducted on 6-core Intel Xeon E5 processor with 24 GB RAM and 1TB HDD available. To take accurate measurements, we measured times directly at the beginning and the end of operator's first-order function with the Java function `System.currentTimeMillis()`. We report the average of three runs for each operator instantiation.

Figure 3.2 shows average execution times and Figure 3.3 shows average startup times for different operator instantiations in logarithmic scale. IE operators are displayed with yellow bars, WA operators are displayed with red bars, and for comparison, we also show the execution time for the `fltr` operator from the Base package. Not surprisingly, the fastest operators in terms of execution time are those performing relatively simple operations, such as splitting text (`splt-txt`), replacing words with their stems (`repl-stem`), or emitting entities (`emit-ent`). Among the most time consuming operators are operators for sentence parsing (`anntt-parse`), entity annotation (`anntt-ent`), and part-of-speech tagging (`anntt-pos-opennlp`). Even different instantiations of the same operator annotating the same type of IE information can differ significantly as the measurements for the `anntt-ent` operator instantiation show. The instantiation `anntt-ent-tr-disease` is faster than `anntt-ent-linnaeus-disease` by a

factor of 40 and the instantiations for gene name recognition, `anntt-ent-banner-gene` and `anntt-ent-linnaeus-gene`, differ by a factor of 74. These differences in measurements can be explained by (a) the time complexity of the underlying algorithms and (b) by the quality of the available library implementation. For example, sentence parsing has a time complexity of $O(n^3)$, entity annotation using automaton matching has a time complexity of $O(n)$, and entity annotation with Conditional Random Fields has a time complexity of $O(n * m^2)$, where n is the length of the input string and m is the number of available states. Moreover, consider the observed runtime differences of the entity annotation of disease and gene names with Linnaeus. Both instantiations are based on the same algorithms with a complexity of $O(n)$ (automaton-based dictionary matching), but differ roughly by a factor of two. The runtime differences are related to the sizes of the dictionaries, which are internally converted into k automatons, where k is the number of dictionary entries and k is larger by a factor of 10 for gene names compared to diseases.

Startup times of non-relational operators are also an important factor of influence for the overall data flow execution time as shown in Figure 3.3. Although startup times only occur once on each processing node per data flow execution, it may notably decelerate data flow execution. In contrast to relational operators, some IE need a substantial amount of time to load dictionaries, models, or automatons before data processing. For example, the automaton-based entity annotation operators `anntt-ent-linnaeus-disease` and `anntt-ent-linnaeus-gene` need between 7.5 (disease) and 33 minutes (gene) for startup. Both dictionaries for genes and diseases contain up to several 100,000 regular expressions, from which matching automatons are assembled, which explains the enormous amount of time needed for startup.

Taken these observations together with the fact that only some IE and WA operators are in a precedence relation with each other emphasizes the importance and the potential of optimizing the execution order of non-relational operators in data flows. This topic will be addressed in Chapters 4 and 5.

3.5 Summary

In this chapter, we introduced operators addressing fundamental IE tasks and WA tasks. We showed how end-users can properly call and configure IE and WA operators in a declarative and user-friendly way in Meteor to enable domain-specific applications using a variety of concrete operator instantiations. Furthermore, we showed how elementary operators can be composed into complex operators to ease the definition of complex analytical tasks on unstructured data and we showed how operators can be adapted to different application domains. In contrast to related work as discussed in Section 2.2, the presented operators allow users to design complex analytical tasks data flows with custom UDFs in a user-friendly manner through declarative Meteor queries inside Stratosphere. Finally, we discussed differences between concrete operator instantiations regarding physical and algebraic properties and characterized operator instantiations with their execution and startup behaviour to highlight both the potential and importance of optimizing the execution order of non-relational operators in parallel data flows. In the following chapters, we show how such data flows can be optimized by analyzing the operator's semantics to improve scalability to huge document collections.

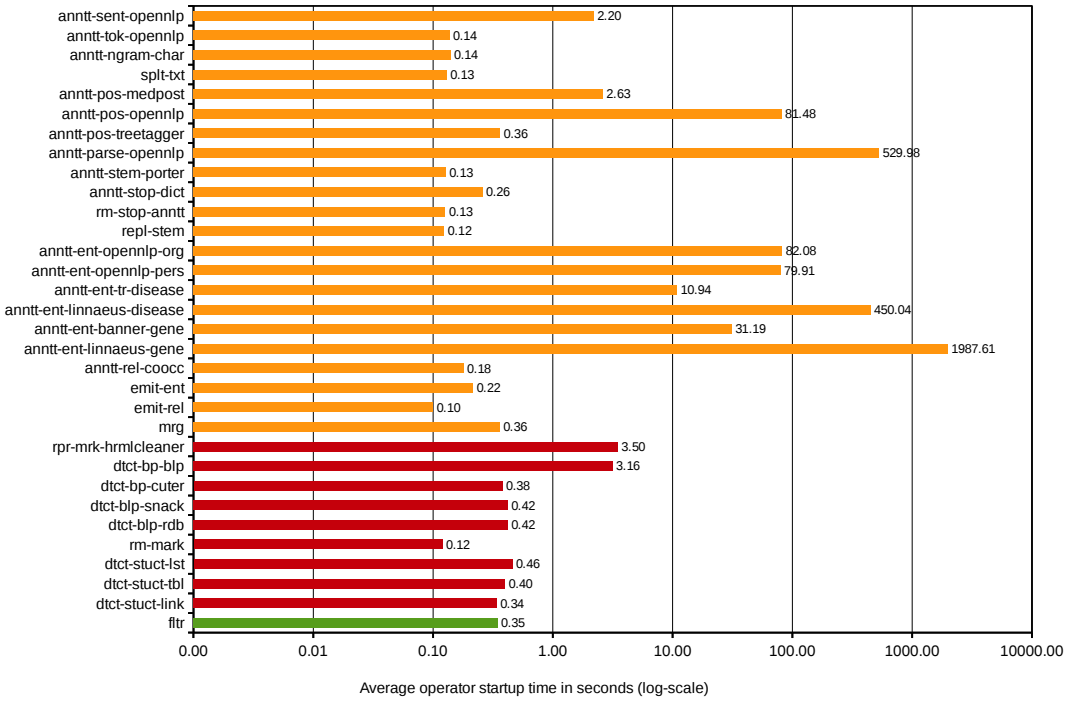


Figure 3.3: Average startup time of IE and WA operators (log-scale). Yellow bars show concrete IE operator instantiations, red bars show concrete WA operator instantiations. For comparison, the average startup time of the Meteor operator `fltr` for filtering records (green bar) is also shown.

4 Optimization of data flows with UDFs: A survey

Many large-scale, domain-specific applications are built upon advanced user-defined functions (UDFs) to enable operations beyond relational algebra [Marx, 2013; Howe et al., 2008; Berriman and Groom, 2011]. As discussed in Chapters 2 and 3, declarative data flow languages are a key component to achieve user-friendliness in parallel data analytics systems, since they (a) allow the expression of complex data flows in form of queries or scripts, (b) reduce implementation efforts for programmers by providing data analytics operators, (c) allow a flexible deployment of data flows on different infrastructures, and (d) enable the adaption of the final execution plan to the properties of concrete data and systems at hand. Apart from Meteor, several other languages have been proposed (e.g., [Thusoo et al., 2009; Olston et al., 2008; Beyer et al., 2011; Alexandrov et al., 2015]), which often provide both relational and non-relational operators to perform arbitrary data analytics tasks.

A main advantage of declarative data flows is the opportunity to optimize and transform these into efficient parallel execution plans through data flow optimizers. However, in contrast to analytical workloads in the relational world where the semantics of operations in terms of optimization is well understood, data flows with UDFs can exhibit various kinds of behavior. Such a behavior is difficult to describe in an abstract, optimization-enabling way and often, optimizers of current data analytics systems disregard UDFs during the optimization process. At the same time, a proper optimization of data flows with UDFs can reduce the execution costs by orders of magnitude [Hueske et al., 2012; Rheinländer et al., 2015].

Research on optimizing non-relational workloads has a long tradition in the database community. In this chapter, we survey techniques for optimizing data flows with UDFs in parallel data analytics systems, which often originate from decades of database research and which apply to different stages of the optimization process. We first review techniques for syntactical data flow transformation followed by a discussion of methods for the analysis of operator semantics to derive precedence relationships and rewrite options between pairs of data flow operations. Third, we study logical and physical data flow transformations. We illustrate each technique using concrete data flow examples and describe the availability of each method in existing systems. Finally, we provide an overview on declarative data flow languages for parallel data analytics systems and we outline optimization techniques available in these systems. Note that this chapter also considers the optimizer SOFA (cf. Chapter 5) as a component for semantics-aware data flow optimization. Here, we only give a brief overview on SOFA, details are explained in Chapter 5.

Figure 4.1 contains a taxonomy of concrete approaches for each optimization phase, which we discuss in the remainder of this chapter, which is structured as follows: The next two paragraphs summarize existing surveys in the area of parallel data analytics

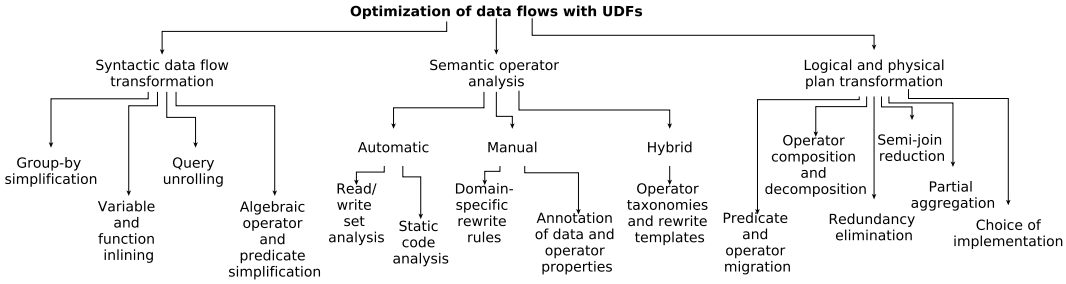


Figure 4.1: Taxonomy of optimization techniques for data flows with UDFs.

systems and we introduce a running example used for explaining concrete techniques for data flow optimization. Section 4.1 discusses techniques for syntactical data flow transformation, i.e., approaches for variable and function in-lining, group-by simplification and query unrolling, and simplification of operators and predicates based on algebraic transformations. A major challenge in data flow optimization is to determine precedence constraints between UDFs, since they are often not part of an algebra with clearly defined rewrite rules. Approaches that address this challenge are presented in Section 4.2. Section 4.3 discusses techniques for data flow transformations on the logical and on the physical level, namely methods for operator composition and decomposition, redundancy elimination, and predicate and operator migration. Optimization of communication costs in the presence of UDFs is also discussed as well as strategies for choosing appropriate operator implementations. Section 4.4 gives an overview of existing data flow languages available for parallel data analytics systems and summarizes key characteristics regarding application areas, targeted processing systems, and optimization techniques. Finally, this survey concludes in Section 4.5.

Previous surveys

Multiple surveys in the area of parallel data analytics systems have been published in the past years, each focussing on different aspects of these systems and complementary topics compared to our work.

One of the first surveys on parallel data processing systems that build upon the Map/Reduce programming model is presented by Lee et al. [2012]. After introducing the Map/Reduce programming model and its variants, the authors discuss advantages and drawbacks of the model, and provide an overview of approaches and strategies enhancing this model. Important application areas (e.g., biomedical data analytics, statistics, data warehousing) and open research challenges are also summarized.

Later, Sakr et al. [2013] survey parallel data analytics systems that build upon the Map/Reduce programming model. Enhancements of the original approach by Dean and Ghemawat are discussed (e.g., iterations, data placement and storage formats, optimization of join processing) and the authors provide an overview on parallel data analytics systems and SQL-like query languages built on top of these systems.

Rumi et al. [2014] survey approaches for task and application scheduling regarding different optimization goals (e.g., data locality, skewness reduction, system utilization) in the Hadoop ecosystem.

Doulkeridis and Nørvåg [2014] provide an overview of Map/Reduce-based data processing techniques followed by an analysis of the most significant limitations of the underlying execution model and suggest a wide range of performance improvements to overcome these limitations. This work also provides a broad and insightful overview of existing approaches for general aspects of data flow optimization (e.g, data access, fair work allocation, non-redundant processing). Yet, optimization of UDFs is not in the focus of this work.

The most comprehensive survey to date is authored by Li et al. [2014] and presents an overview of the Map/Reduce programming model, systems implementing this model, extensions and enhancements compared to the original proposal, as well as languages for specifying analytical data flows for database-style applications. The survey focuses on approaches for implementation, scheduling, and optimization of relational operators using the Map/Reduce programming model.

Recently, Bajaber et al. [2016] review the state-of-the-art in parallel data processing systems for huge data sets. Discussed systems are categorized in a taxonomy and distinguished based on their application domain, i.e., systems for general purpose, database-style, graph analytics, and stream processing.

By focussing on the processing and optimization of data flows with UDFs in parallel data analytics systems, this chapter complements the perspectives on parallel data processing presented in the previous surveys.

Running example

In the following sections, we introduce a multitude of different optimization techniques and show their potential for optimizing data flows with UDFs. We explain their premises by means of an example query script formulated in Meteor and its corresponding data flow. Whenever necessary, we slightly modify and extend this example to clarify the benefits of certain optimizations.

Our example query analyzes two dumps of Wikipedia articles gathered at different points in time to determine NASDAQ-listed companies, which went bankrupt within a certain time frame, together with an investigation of persons related to these companies. The corresponding logical data flow is shown in Figure 4.2, the concrete Meteor query is contained in Listing 4.1. We added comments (i.e., text starting with `'//'` in Listing 4.1) to precisely identify operators occurring multiple times in the depicted data flow. The data flow is DAG-shaped and consists of 12 complex operators, which can be expanded to 22 elementary operators, 13 of which are non-relational.

The first line of the query imports a package of non-relational operators that perform tasks in the areas of IE and NLP. Lines 3–11 contain the ad hoc definition of a complex UDF consisting of several IE operators for sentence and token annotation, for annotating multiple entities in texts (i.e., organizations, persons, and dates), and for extracting relationships between persons and organizations. The data sets to be analyzed are retrieved from the file system in Lines 13 and 14. Analysis of both data sets is carried out through calling the UDF `ie_pipeline` and by a subsequent filter operation (Lines 16–18 for data set "new" and Lines 20–22 for data set "old", respectively). The newer data set is filtered for articles that contain the term "bankrupt" (Line 18) and the older data set is filtered for articles that contain the term "NASDAQ" but do not contain the term

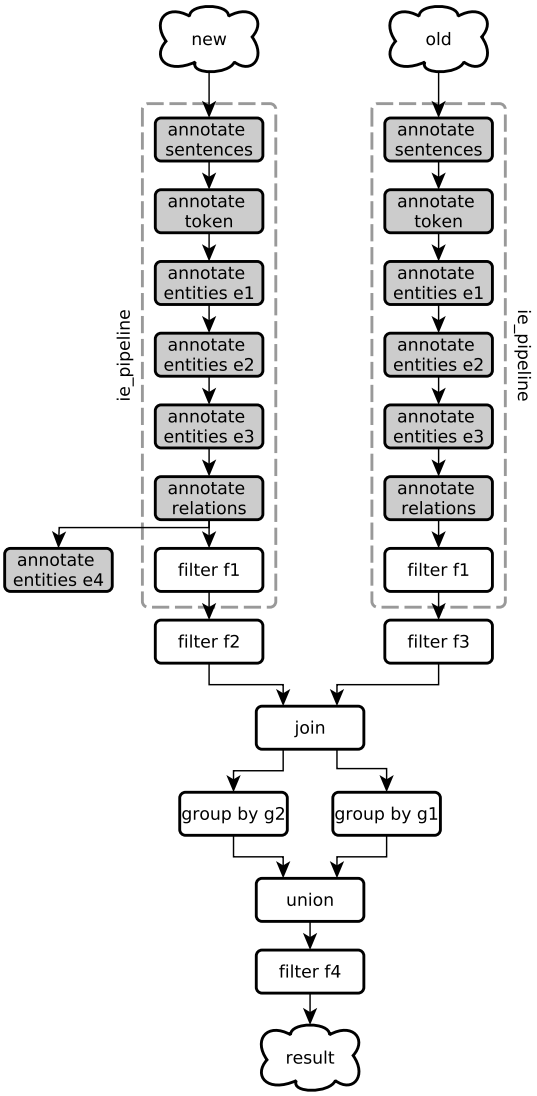


Figure 4.2: Data flow for Meteor query shown in Listing 1. UDFs are colored in grey.

Listing 4.1: Meteor query for advanced information extraction combining UDFs, non-relational, and relational operators.

```
using ie;

ie_pipeline = fn(input){
  input = annotate sentences input;
  input = annotate tokens input;
  input = annotate entities input type "organization"; //e1
  input = annotate entities input type "date"; //e2
  input = annotate entities input type "person"; //e3
  input = annotate relations input type ["organization","person"];
  input = filter $article in input where ($article.annotation_relation); //f1
}

$new = read from 'file:///new_articles/';
$old = read from 'file:///old_articles/';

$a = ie_pipeline($new);
$loc =  annotate entities $a type "location"; //e4
$a = filter $article in $a where (substr($article.text,"bankrupt")>=0); //f2

$b = ie_pipeline($old);
$b = filter $article in $b where (substr($article.text,"NASDAQ")>=0) and
    (substr($article.text,"bankrupt")<0); //f3

$joined = join $a, $b where ($a.id==$b.id);

$results_year = group $a in $joined by $a.annotation_entity.date.year into {
       $a[0].annotation_entity.date.year,
       rels = $a[*].annotation_relation
    }; //g1

$results_monthyear = group $a in $joined by $a.annotation_entity.date.year and
    $a.annotation_entity.date.month into {
        $a[0].annotation_entity.date.year,
        $a[0].annotation_entity.date.month,
        rels = $a[*].annotation_relation
    }; //g2

$results = union $results_year, $results_monthyear;

$results = filter $r in $results where (($r.year>=2014 and $r.year<2016) or
    ($r.year>=2014 and $r.year<2016 and $r.month>6)); //f4

write $results to 'file:///result';
```

"bankrupt" (Lines 21–22). Additionally, locations are annotated in the data set "new" (cf. Line 17).

Both data sets are joined on the document ID to retain only those documents where companies are described as bankrupt in the newer data set but were not described as such in the older data set (Line 24). Afterwards, two grouping and one union operator (cf. Lines 26–38) are applied to group the joined data by year and month (Lines 31–36) and also by year only (Lines 26–29) to account for incomplete date annotations. Unnecessary attributes of the grouped data are projected out by using the *into* keyword. The result set is filtered for companies which went bankrupt between 2014 and 2016 (Lines 40–41), and is finally written to the file system (Line 43).

4.1 Syntactic data flow transformation

The transformation of data flows on the syntactic level is applied as a first step within the optimization process (cf. Chapter 2). Given a query script, which describes a data flow with UDFs, the following techniques may be applied:

1. rule-based in-lining of variables and functions,
2. query unrolling and group-by simplifications, and
3. operator and predicate simplifications.

The goal of syntactic transformation is not only to reduce complexity within the query script, but also to discover potential for inter-operator parallelism of contained operators. Although the techniques discussed below are not specific to data flows that contain UDFs, the presented transformations greatly impact the effectiveness of downstream logical optimization in such flows. By first applying operator simplification and query unrolling, complex UDFs are translated into concrete data flow operators, which enables advanced data flow transformations in succeeding steps of the optimization process (cf. Sections 4.2 and 4.3).

4.1.1 Rule-based variable and function in-lining

Variable and function in-lining replaces a variable or function call with the variable's value or the function body, similar to macro expansion known from procedural programming and scripting languages. Variables can be in-lined if they are referenced only once in expressions or function calls of a query. Similarly, calls of UDFs can be expanded with the function's body if the function's call-tree is not recursive. In these cases, parameters are replaced by local variables, which are possibly in-lined to further simplify the query.

Function in-lining is important as an enabler for downstream plan transformations that are not possible without. Consider Listing 4.2, which contains an excerpt of the query from Listing 4.1. Lines 3–11 of the script declare a UDF called `ie_pipeline` to analyze texts. Internally, this UDF calls six other UDFs (Lines 4–9) and one filter operator (Line 10). When translating the complete query without UDF expansion, the resulting data flow consists only of two consecutive operators, `ie_pieline` and `filter`, whereas it consists of eight operators after expanding `ie_pipeline` as shown in Figure 4.3.

Listing 4.2: Excerpt of Meteor query from Listing 4.1, UDF expansion.

```
...
ie_pipeline = fn(input){
  input = annotate sentences input;
  input = annotate token input;
  input = annotate entities input type "organization"; //e1
  input = annotate entities input type "date"; //e2
  input = annotate entities input type "person"; //e3
  input = annotate relations input type ["organization","person"];
  input = filter $article in input where ($article.annotation_relation);//f1
}
...
$old =  read from 'file:///old_articles/';
...
$b = ie_pipeline($old);
$b = filter $article in $b where (substr($article.text,"NASDAQ")>=0) and
  (substr($article.text,"bankrupt")<0); //f3
...
```

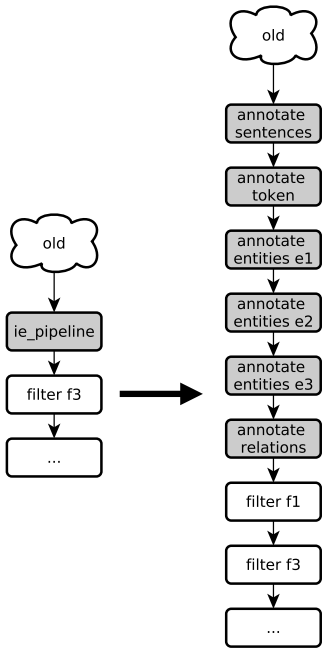


Figure 4.3: Expansion of UDFs during query simplification. UDFs are colored in grey.

Listing 4.3: Excerpt of Meteor query from Listing 4.1, optimization of group-by.

```
...
26 $joined = filter $a in $joined where $a.annotation_entity.date.month == 'June';
27 $results_year = group $a in $joined by $a.annotation_entity.date.year;
28 $results_monthyear = group $a in $joined by $a.annotation_entity.date.year and
29   $a.annotation_entity.date.month;
...
```

In principle, the expanded data flow has six possible operator execution orders of the `annotate entities` operators that are semantically equivalent to the initial query (cf. Chapter 5). Moreover, the expanded data flow also contains two adjacent filter operators, which can be reordered with any database-style optimizer.

Variable and function in-lining is available in some optimizers for parallel data analytics systems [Beyer et al., 2011; Murray et al., 2011], yet, the direct benefits of in-lining have not been evaluated systematically in any system. It is known from programming languages research that in-lining depending on the language yields speed-ups between 10% and 40% at the cost of slightly increased space requirements [Santos, 1995; Wörteler et al., 2015].

4.1.2 Group-by simplification

Computing aggregates and groupings is one of the most frequent operation in data analytics applications that involve UDFs, since very often, results computed by the UDFs need to be aggregated for interpretation by human data analysts. Often, the grouping functions contain expensive user-defined predicates [Jacobs, 2009] and optimization of such operations is crucial for the overall performance of data analytics programs. Next to logical and physical data flow transformations regarding aggregations and groupings, which are discussed in Section 4.3, simplification of such operations is also a powerful technique.

Group-by operations can be simplified by eliminating redundant grouping columns in cases where "equals" selection predicates are specified on them. Consider Listing 4.3, which modifies the query from Listing 4.1 in Line 26 by adding a filter operator, which retains only records where the month June is annotated. Lines 27–29 group the data set by year and month, the definition of the output schema is omitted for brevity in this example. Since Line 26 only produces results for a certain month, the group-by operation in Lines 27–29 can therefore be altered as shown in Listing 4.4. This simplification reduces the number of key columns to be considered and simplifies aggregation operations performed later in a data flow. Reduction of grouping columns is applied in relational database systems, e.g., in IBM DB2[15] and SQL Server [Simmen et al., 1996], yet we are not aware of any data flow optimizer currently employing this technique.

Another simplification technique for group-by operations in data flows, called *grouping sets*, applies to situations where multiple group-by operators process the same grouping keys. Grouping sets are logically equivalent to expressing several `group by`

[15] IBM DB2 for z/OS Technical Overview, `https://www.redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/sg248180.html`, last accessed: 2016-10-31.

Listing 4.4: Excerpt of Meteor query from Listing 4.1, elimination of redundant grouping attributes.

```
...
$results_monthyear = group $a in $joined by $a.annotation_entity.date.year into {
      $a[0].annotation_entity.date.year,
      $a[0].annotation_entity.date.month,
      rels = $a[*].annotation_relation
    };
...
```

operations and connecting them with `union` operations. For example, Lines 26–36 of Listing 4.1 can be rewritten into the statement shown in Listing 4.5.

Listing 4.5: Excerpt of Meteor query from Listing 4.1, grouping sets optimization.

```
...
$results_monthyear = group $a in $joined by $a.annotation_entity.date.year as y and
    $a.annotation_entity.date.month as m grouping sets ((y, m), y) into {
        $a[0].annotation_entity.date.year,
        $a[0].annotation_entity.date.month,
        rels = $a[*].annotation_relation
    };
...
```

This rewriting is beneficial in many cases, since it collapses three operators into a single one that evaluates all grouping predicates. Grouping set operations are well-known from data warehousing systems [Gray et al., 1997] and they are also available in the parallel data processing system Hive [Thusoo et al., 2009].

4.1.3 Query unrolling

The unrolling of nested queries is important for optimizing data flows with UDFs, since nested queries involving UDFs are prone to cause high memory and CPU loads during query processing[16]. Sub-queries can be distinguished into uncorrelated and correlated sub-queries. In correlated sub-queries, the nested query references one or more data sets processed by the outer part of the query. Optimization of such queries has been addressed since the early ages of relational query optimization research. A classification of nested queries was first introduced by Kim [1982], based on which different techniques for unnesting were developed by Ganski and Wong [1987].

Uncorrelated sub-queries are independent of the outer query and can therefore be optimized and executed separately, whereas optimization of correlated sub-queries is significantly more difficult and often leads to inferior plans [Neumann, 2009]. Optimizers typically unroll the nested part of the correlated query as much as possible to evaluate the nested part separately and combine it with the outer part by an explicit join operation [Selinger et al., 1979]. This allows the data flow optimizer to identify more efficient rewrite options, for example, by pushing projections or selections from the outer to the inner part of the query.

[16] For example, consider bug reports of Hive `https://issues.apache.org/jira/browse/HIVE-372`, `https://issues.apache.org/jira/browse/HIVE-5494`, both last accessed: 2016-10-31.

Listing 4.6 shows an excerpt of Listing 4.1, which is modified with a correlated sub-query filtering the join results for all years where at least 10 bankruptcies of NASDAQ-listed companies were found. The nested part of the query is connected to the filter operation in Line 26. It starts with first isolating date annotations from the remaining annotations (Lines 27–30), which are subsequently transformed into a normal form by a user-defined operator `normalize` (Line 31). Normalized dates are grouped by year and occurrences of years are counted (Lines 32–36). Finally, the grouped data is filtered for those years occurring at least 10 times (Line 37). After computing the correlated sub-query, the data flow continues unaltered.

Listing 4.6: Correlated sub-query inspired by Listing 4.1.

```
...
24 $joined = join $a, $b where ($a.id==$b.id);
25
26 $joined = filter $a in $joined as $results where $a.id in (
27       $date = transform $a in $results into {
28         id = $a.id,
29         date = $a.annotation_entity.date
30       };
31       $date = normalize $a in $date type 'date';
32       $years = group $y in $date by $y.date.year into {
33         year = $y[0].date.year,
34         count = count($y[*].date.year),
35         id = toArray($y[*])
36       };
37       $years = filter $y in $years where $y.count>10 into {id = $y.id};
38 )
39 $joined = filter $a in $joined where $a.annotation_entity.date.month == 'June';
...
```

Since the nested part of the query is used as a filter predicate, it needs to be computed prior to the actual execution of the outer part and no rewrite options can be applied immediately. By unrolling Lines 27–38 of Listing 4.6, the partial data flow depicted in Figure 4.4 is created and optimizations can be applied, for example, by reordering the filter operator (cf. Line 39) as indicated by the grey dashed arrows. To combine the results of the nested part of the query with the remaining outer part, an explicit outer join operator is introduced into the data flow.

Nested sub-queries are available for example in Impala [Kornacker et al., 2015], which allows both correlated and uncorrelated sub-queries in the WHERE clause, and rewrites these into join queries. The Scope data flow optimizer [Chaiken et al., 2008] optimizes correlated sub-queries by applying combinations of outer and semi-joins together with user-defined combiners.

4.1.4 Algebraic data flow and predicate simplification

UDFs are often configured with complex predicates, which greatly impact the performance of the UDFs itself and the overall data flow. To combine multiple predicates, boolean expressions are applied, which might contain redundancies and and a naïve evaluation of such predicates leads to unnecessary computation.

Simplification of redundant predicates involves rewritings based on the well-known boolean algebra, which were summarized for relational query optimization by Özsu and

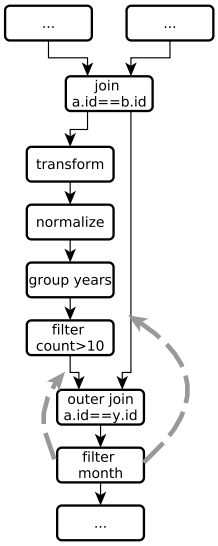


Figure 4.4: Unrolled correlated sub-query from Listing 4.6. Grey dashed arrows indicate reorder options in the unrolled query.

Valduriez [1999]. Some of the transformation rules can be applied in any situation, such as the removal of double negation. For example, the filter predicate applied in Lines 40–41 of the Meteor query shown in Listing 4.1 can be simplified using laws of the boolean algebra into the statement shown in Listing 4.7.

Listing 4.7: Excerpt of Meteor query from running example, algebraic predicate simplification.

```
...
40 $results = filter $r in $results where ($r.year>=2014 and $r.year<2016);
...
```

Other rules directly affect the execution order of predicates, for example, consider two predicates $p1$ and $p2$, which are combined with the boolean `or` operator ($\vee$). Since $\vee$ is commutative (i.e., executing $p_1 \vee p_2$ is equivalent to executing $p_2 \vee p_1$), any execution order of $p1$ and $p2$ is correct but yields different costs [Minker and Minker, 1980]. Therefore, each such rewrite should be based on cost estimates or a combination of cost estimates and probability values for predicates p_1 and p_2 [Hanani, 1977; Brown, 1998]. Rewriting should also ensure that semantics of boolean predicates is preserved, for example, the semantics of a predicate $p_1 \&\& p_2$ is defined as "first execute p_1 followed by p_2" and should thus not be rewritten to $p_2 \&\& p_1$. Simplification of boolean expressions is available in many compilers for data flow languages, for example in Pig [Olston et al., 2008], Hive [Thusoo et al., 2009], or Spark SQL [Zaharia et al., 2010].

Removal of idempotent UDFs is particularly important for data flow simplification. We call a unary UDF o *idempotent* if it produces the same result regardless how often it is applied to some input I: $o(o(I)) \equiv o(I)$. N-ary UDFs are idempotent if all its inputs are idempotent with respect to the function applied to them, i.e., when applied to n equal values, an idempotent function returns the value as result. For example, a function that determines the minimum value of two equal inputs is idempotent: $min(I_1, I_1) = I_1$ and

a UDF `mrg`, which merges text annotations, is also idempotent. Eliminating multiple calls to idempotent UDFs from a data flow before its execution is always beneficial, for example, selections and projections on the same predicates are idempotent [Özsu and Valduriez, 1999]. For UDFs, idempotency needs to be annotated or derived to eliminate redundant executions of o. This technique is for instance applied in SOFA (cf. Chapter 5, [Rheinländer et al., 2015]).

4.2 Semantic analysis of UDFs

To enable data flow rewritings that involve reordering, introduction, or removal of operators, UDF meta data (e.g., to describe selectivity estimates or semantic properties like associativities or commutativities) needs to be available to the optimizer. Accurately describing the semantics of UDFs in terms of optimization is one of the most persistent challenges for data flow optimization, since UDFs exhibit all sorts of behavior, which is difficult to describe in a general, optimizer-friendly way. In contrast to relational operators, whose semantics is well understood [Özsu and Valduriez, 1999], UDFs have different properties than the classical relational ones and finding the right set of properties for describing UDFs in terms of optimization is not trivial. Moreover, optimizing data flows with UDFs requires novel approaches for data flow transformations, which are usually captured in concrete rewrite rules or described by precedence constraints contained in a data flow. In the following, we distinguish three classes of approaches that address the problem of describing the semantics of UDFs, namely

1. approaches that annotate UDF manually,
2. approaches that perform automated code analysis to infer semantics, and
3. approaches that combine (1) and (2).

4.2.1 Annotation of UDF semantics

Manually annotating UDFs with properties that describe their semantics, their behavior, and defining concrete rewrite rules for UDFs requires a deep knowledge of the application domain. Domain- and application-specific rewrite rules are available for different areas, such as for IE data flows, ETL flows, or scientific workflows. In the following, we briefly summarize some of these domain-specific approaches.

Wachsmuth et al. [2011] define rules for rewriting information extraction data flows based on domain-specific knowledge. These rules include decomposition of extraction tasks, early filtering, and lazy evaluation. User-defined extraction tasks are distinguished into required and optional tasks and optional tasks are only executed if needed to satisfy a concrete information need. For example, an extraction data flow as shown on the left of Figure 4.3, is decomposed into its components as shown on the right side of this figure and each task is examined whether it is needed to produce the output result. In this example, all tasks are needed and thus included in the execution plan for the shown data flow.

Ogasawara et al. [2011] propose an algebraic approach for task definition for the scientific workflow system Chiron [Ogasawara et al., 2013] to optimize scientific workflows with many UDFs. UDFs are manually assigned to a fixed set of six operations

(e.g., Map, Reduce, Filter) based on the ratio of consumption and production between input and output records. For example, the `anntt-ent` UDFs from Figure 4.2 would be assigned to *Map* since they produce and consume exactly one record. Similarly, a `group by` UDF would be mapped to *Reduce*, since it groups a set of n records into a single record based on a given grouping attribute. By applying these assignments, UDFs obtain a clear semantics and can be optimized through algebraic transformations and by analyzing producer-consumer dependencies between operators.

Simitsis et al. [2005] reason about manually annotated properties of ETL flows such as schema information, operator adjacency in ETL processes, or the number of consumers and producers to determine possible optimizations.

Carman Jr. et al. [2015] provide rewrite rules for processing XML documents with XQuery on top of Algebricks [Borkar et al., 2015] and the parallel data analytics system Hyracks [Borkar et al., 2011]. Rules for rewriting path expressions attempt to remove redundant parts of the data flows introduced by unnesting operations (cf. Section 4.3). Further rewrite rules perform plan transformations to enable a parallel computation of certain XQuery constructs (e.g., data access, aggregation, join).

Kougka and Gounaris [2013] annotate and analyze binding patterns to determine precedence constraints between operators in data flows. This technique is derived from query optimization in the context of web services [Srivastava et al., 2006]. First, a global schema is defined for the data flow and annotations of produced and consumed attributes are created. Consider Figure 4.5, which shows a global schema for the `ie_pipeline` of Listing 4.1 (cf. Lines 3–11) together with information on produced and consumed attributes. Recall that precedence constraints exist between any two operators a, b if a produces attributes consumed by b or vice versa. In our example from Figure 4.5, precedence constraints exist for example between sentence and token annotation, and between entity and relationship annotation. The only operator, which can be placed at any position of `ie_pipeline`, is `filter f2`, since is not in a precedence relationship with any other operator.

To improve optimization of data flows with UDFs, Spark, AsterixDB, and Stratosphere allow to manually annotate and provide rewrite rules for UDFs. In the following section, we discuss methods to automatically derive UDF semantics, such as producer-consumer relationships, through code analysis.

4.2.2 Inference of UDF semantics through code analysis

Inferring the semantics of UDFs and precedence relationships between pairs of UDFs automatically without manual annotations provided by the user requires techniques that deeply analyze the UDF's program code without executing it.

Hueske et al. [2012] show that analyzing a few properties of UDFs suffice to enable semantically equivalent reordering of user-defined selections, joins, and certain aggregations without having access to UDF annotations. Particularly, they define read and write sets, which are similar to the notion of producers and consumers. Provided that complete information on read/write sets is available, two consecutive record-at-a-time UDFs A, B (e.g., filter, projection, map,...) can be safely reordered if no read/write conflicts between A and B exist, i.e., if A does not access attributes modified by B and A does not modify attributes modified or accessed by B and vice versa. Similarly, key-at-a-time UDFs (e.g., reduce, aggregations, joins) can be reordered if they have no read/write

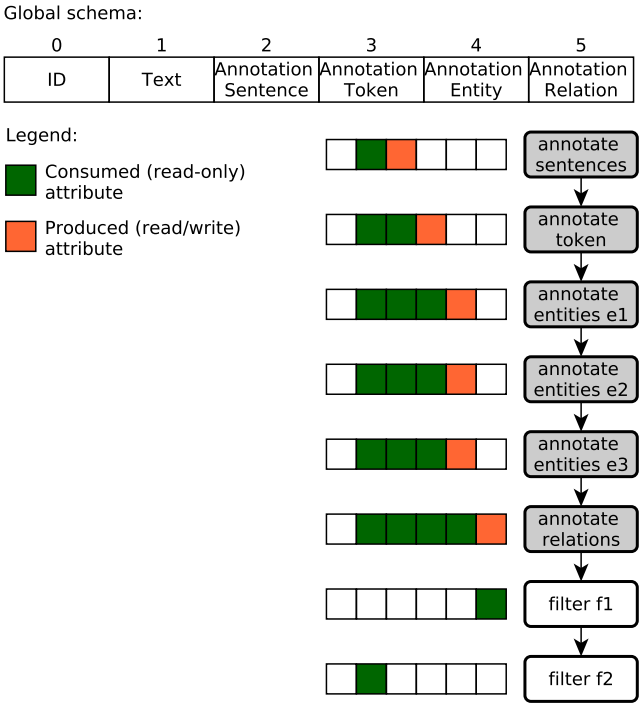


Figure 4.5: Global schema and read/write set information for `ie_pipeline` sub-flow. Orange boxes indicate for each UDF produced (written) attributes and green boxes indicate consumed (read) attributes. UDFs are colored in grey.

conflicts and if either all or no records with a specific grouping key k are removed during reordering (so-called *key group preservation*). Read/write sets can either be annotated manually or be obtained from static code analysis [Hueske et al., 2013].

To enable code analysis, a given data flow is first analyzed to infer a global schema and each data flow operator is translated into three-address code (TAC). Figure 4.5 displays a global schema for the UDF `ie_pipeline` from Listing 4.1 together with annotations of read and written attributes. Green marked attributes in Figure 4.5 correspond to the read set of each UDF and orange marked attribute constitute the write set of each UDF. Listing 4.8 shows the TAC of the filter $f2$ from our running example. The read set is determined by scanning the UDF's code for statements of the form `$t:=getField($i,n)`, which assign the content of attribute n of the input record i to a temporary variable `$t`, and all subsequent uses of `$t` in the code. If `$t` is used, it is added to the read set of the UDF. In Listing 4.8, Line 2 contains a `getField` statement, which assigns the content of the attribute at index position 1 of the input record to a temporary variable `$a`. Subsequently, `$a` is used to perform the actual filter operation (cf. Line 3). Since no other attribute of the input record is accessed, the read set only consists of this attribute.

In the same way, static code analysis detects explicit operations that copy, project, modify, or add single attributes to the output to infer the write set of a UDF. Here, the static code analysis algorithm starts to collect statements of the form `emit($o)`, which emit the output record of a UDF, and tracks the origin of `$o` to identify column projections. Remaining attributes of the write set are identified by collecting statements

Listing 4.8: Three address code for filter f2 used for static code analysis.

```
1 f2(InputRecord $i)
2 $a:=getField($i,1)
3 if(not contains($a, 'bankrupt')) goto 6
4 $o:=copy($i)
5 emit($o)
6 return
```

of the form `setField($o,n,$a)` from the TAC of a UDF. In our example from Listing 4.8, the write set is empty since no attribute is modified.

Guo et al. [2012] apply code analysis not only to determine precedence relationships between UDFs to enable code motion and early filtering, but also to identify options for reducing attributes automatically. To perform attribute reduction, the set of output attributes, which is used by subsequent UDFs in the data flow topology is determined. Unused output attributes and UDFs, which solely contribute to computing the removed attributes, are removed. For example, in Figure 4.5 the attribute at index position 0 can be removed, because it is not accessed by any of the UDFs. Fan et al. [2015] present an approach for optimizing the placement of filters inside the body of a UDF using static code analysis. Static code analysis as carried out in Manimal [Cafarella and Ré, 2010; Jahani et al., 2011] also does not consider UDF semantics, but statically analyzes the code of Map/Reduce programs to identify selections, projections, and options for data compression. Static code analysis as presented by Hueske et al. [2012] is implemented in Stratosphere, Manimal [Cafarella and Ré, 2010; Jahani et al., 2011] is available for Hadoop, and techniques of Fan et al. [2015] and Guo et al. [2012] are implemented in the Scope system.

4.2.3 Hybrid approaches

Although static code analysis already enables quite a few optimizations for data flows with UDFs, including semantic information on the UDFs behaviour and concrete rewrite rules for UDFs into the optimization process is crucial to enable advanced data flow transformations. Annotating semantic information and rewrite rules for many combinations of UDFs, however, is time-consuming and requires a deep knowledge of the application domain. Therefore, hybrid approaches, which combine manual annotations with automated UDF analysis and static code analysis aim at comprehensive optimization of data flows with UDFs. To minimize annotation effort, we propose in this thesis to arrange UDFs and properties relevant to data flow optimization (e.g., parallelization function, number of input and output channels, algebraic properties) in taxonomies [Rheinländer et al., 2015]). Relationships within and between those taxonomies concisely describe the semantics of related UDFs and rewriting of UDFs is carried out using a set of rewrite templates, which build upon UDF properties and abstract UDFs. A detailed description of this approach, its implementation in Stratosphere and an evaluation on a diverse set of data flows with UDFs is presented in Chapter 5.

4.3 Optimization by data flow transformation

In this section, we introduce techniques for data flow transformations both on the logical and on the physical level, which can be applied after a data flow has been transformed syntactically, contained UDFs have been analyzed regarding their semantics, and existing precedence relationships and rewrite options have been identified. Logical data flow transformation optimizes the logical execution order of data flow operators while respecting precedence constraints. Physical data flow transformation optimizes concrete execution strategies of a data flow, for example, by choosing appropriate operator implementations depending on the data to be analyzed and the execution environment, by optimizing data transfer across compute nodes, or by optimizing the execution of operators through introducing intermediate caching and indexes. In the following, we review techniques for logical and physical transformation, which we consider most promising for data flows with UDFs, namely

1. operator composition and decomposition,
2. redundancy elimination,
3. predicate and operator migration,
4. partial aggregations,
5. semi-join reducers, and
6. choice of operator implementation.

4.3.1 Operator composition and decomposition

Recall that complex operators are abstractions elementary operators, which are, for example, non-relational operators for annotating texts or for parsing XML documents, or relational operators such as filter, join, or group by. In contrast to UDFs defined ad hoc as employed in Listing 4.1, complex operators are directly embedded into data flow languages and provide short-cuts to adding multiple elementary operators to a data flow [Rheinländer et al., 2015; Simitsis et al., 2012]. Another fundamental difference is that complex operators have a clearly defined semantics, which can either be inferred by an optimizer through code analysis or by evaluating provided operator annotations.

Decomposing complex operators into contained elements is important for optimizing data flows with UDFs, since complex operators exhibit different semantics than their elementary components in many cases (cf. Chapter 3). Furthermore, these operators are mostly designed to address non-relational needs, although some contained parts may be of relational origin (e.g., filter, projection). For example, consider a complex operator `tokenize` for splitting text into separate words. This UDF internally consists of two elementary operators, `annotate tokens` and `split tokens`. The `annotate tokens` operator recognizes start end positions of individual words in text by analyzing white spaces and other delimiters. The `split tokens` operator splits the text into separate words based on the boundaries detected by `annotate tokens`. `Tokenize` and its elementary components differ in terms of prerequisites regarding the properties of the input data (i.e., existing sentence boundary annotations), accessed and modified attributes of the input data, and the ratio between input (I) and output (O) data as shown

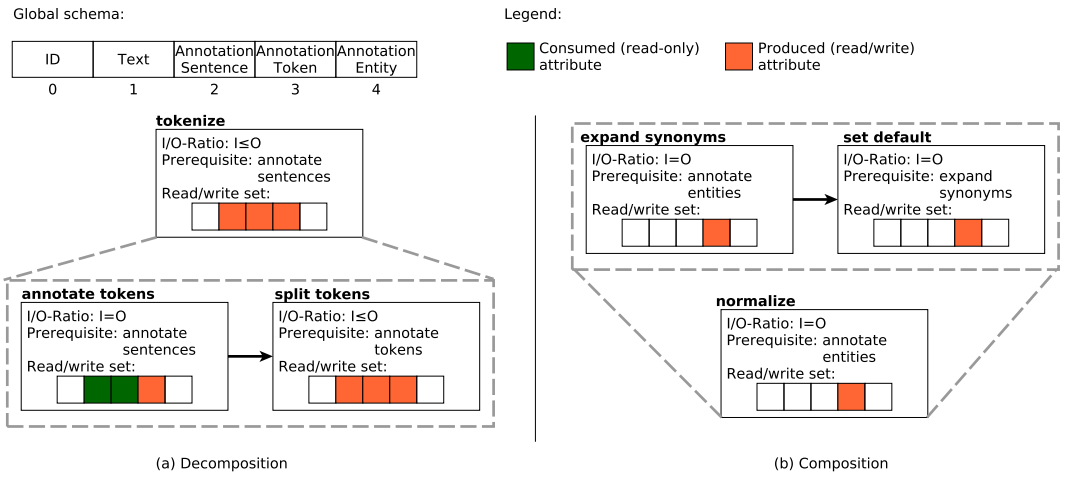


Figure 4.6: Composition (right) and decomposition (left) of complex, user-defined operators.

in the left part of Figure 4.6. The operator `annotate tokens` only produces information on existing token boundaries in the input, while `split tokens` actually splits the input data into multiple smaller units. This yields many output records for each analyzed text depending on the number of identified token boundaries. Thus, executing the `split tokens` operator as late as possible is highly beneficial in many complex analytical flows. Such a rewrite option is not detectable without decomposing `tokenize` into its elementary components first, since the global behavior of `tokenize` conforms to the `split tokens` operation, where text is analyzed and in turn is split up into individual words.

In other cases, the reverse operation only enables logical operator reorderings, i.e., composing multiple elementary operators into a single, complex UDF. Consider the right part of Figure 4.6, which displays two text processing UDFs, `expand synonyms` and `set default`, in the upper part. The UDF `expand synonyms` augments existing entity annotations with synonyms from a dictionary and `set default` selects one of these synonyms as representative and removes all others. Reordering these two UDFs is limited, since both modify existing entity annotations. However, a complex UDF `normalize entity` as shown in the bottom of the figure is potentially reorderable. By composing `expand synonyms` and `set default`, modifications of entity annotations only affect internals of existing entity annotation and the number of records and annotations remains unchanged.

During optimization on the logical level, data flows with UDFs should therefore be examined for contained complex operators, which are decomposed into elementary components, and the rewritten data flow should be analyzed again to detect further reorder options. Similarly, consecutive elementary operators in a data flow should be analyzed whether a composition of these yields beneficial reorderings. Operator decomposition is available as part of the SOFA optimizer [Rheinländer et al., 2015] for Stratosphere (cf. Chapter 5). Stubby [Lim et al., 2012] optimizes Hadoop data flows by leveraging manual annotations of Map and Reduce functions to merge multiple Map/Reduce jobs. Simitsis

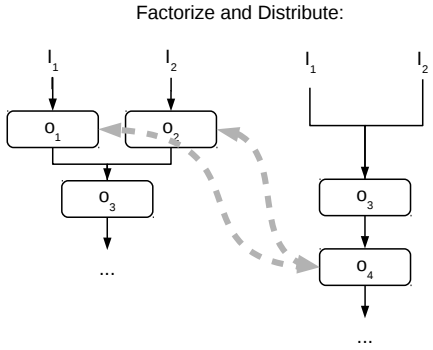


Figure 4.7: Factorize and distribute of user-defined operators.

et al. [2005] introduce a rather general approach called *factorize and distribute* for optimizing operators with two inputs by composition and decomposition in the context of ETL data flows. The left part of Figure 4.7 displays a partial data flow consisting of a dual-input operator o_3 together with two single-input operators o_1, o_2. The operators o_1, o_2 have the same semantics and functionality, but are applied to different inputs I_1, I_2 before executing o_3. Assuming that o_1 and o_2 are not in a precedence relation with o_3, the data flow can be rewritten in the following way: o_1 and o_2 are removed from the graph, the dual-input operator o_3 is applied to I_1 and I_2 to produce an intermediate data set I'. A new operator o_4, which combines the functionality of o_1 and o_2, processes I' and is inserted after o_3. Note that this transformation can also be applied in the reverse direction by replacing o_4 with o_1 and o_2.

4.3.2 Redundancy elimination

Redundancy is often introduced by view resolution and similar data access macros, i.e., the view defines more than is required for the data flow at hand. Redundancy elimination is particularly important for data flows with UDFs, since these flows often contain many compute-intensive UDFs. We consider operators to be redundant if results produced by such an operator are never consumed by any other operator or data sink contained in a data flow, i.e., data produced by such an operator does not contribute to the final result set in any form. Such operators can be safely removed to (a) shrink the size of the data flow to remove complexity from downstream optimization and scheduling phases and, more importantly, (b) to reduce the overall execution time.

Dead code elimination originally emerged from compiler optimization and a classical approach was introduced by Cytron et al. [1991]. In this approach, a data flow is transformed into an intermediate representation called static single assignment (*SSA*). SSA requires that (1) all intermediate data sets are identified by variables, which are properly defined before usage, and (2) that every variable is assigned only once in a data flow to create explicit representations of producer-consumer chains. This requirement can be achieved by versioning existing data flow variables.

Necessary and redundant operators can now be identified as follows: First, all data sinks are added to a queue *working list*. An algorithm for analyzing the critical path of a data flow recursively extracts one item k from the working list and marks it as required

for the data flow. All operators, which have not yet been marked as required and which produce an output that is consumed by *k*, are added to the working list. The algorithm recursively continues until the working list is empty. All remaining operators, which have not been marked as required yet do not contribute to any result set of the data flow and can therefore be safely removed from the optimized data flow.

Consider Figure 4.8, where each operator from Listing 4.1 is assigned a variable in SSA, shown in yellow cornered boxes beneath each operator. The critical operator path is also shown and is marked with red arrows. Individual steps conducted during critical path analysis are shown on the right side of the figure bottom-up. Analysis starts by adding the variable *sink* to the working list (cf. 1st step), which is subsequently marked as required and removed from the working list. At the same time, all variables identifying operators that produce a result set consumed by *sink* are added to the working list. In our example, variable *r2* is added to the working list (cf. 2nd step). The algorithm continues with this procedure until all sources are added to the set of required operators in step 35 and the working list is empty. One operator, which is identified by the SSA variable *loc*, remains unmarked after all sources were processed. This operator is unnecessary to compute any result set and can therefore be removed from the data flow.

In complex analytical data flows, common sub-expressions occur frequently, where intermediate results are processed in many different ways (e.g., joined, aggregated, transformed) and a repeated execution of such common expressions is dispensable. Silva et al. [2012] propose an extension to the Scope optimizer, leveraging query fingerprints and physical properties of shared groups to identify and optimize the execution of common sub-expression in distributed settings.

In parallel data analytics systems, redundant operator removal is carried out by using code analysis similar to SSA as available in SCOPE and Apache Flink [Fan et al., 2015; Alexandrov et al., 2015]. Alternatively, redundant operator removal can be performed by applying rewrite rules based on operator semantics [Rheinländer et al., 2015], or by a structural analysis of the data flow graph [Heise et al., 2012] as available in Stratosphere.

4.3.3 Predicate and operator migration

A fundamental difference between optimization of relational queries and data flows with UDFs is the presence of expensive predicates, which are executed in the latter case often in combination with non-relational or filter operations and are applied to unstructured or semi-structured data. Consider a filter operation, which filters a large collection of XML documents based on XPath predicates. To evaluate the filter condition, both the XPath expression and all documents need to be parsed and compared to each other, which is often expensive [Gottlob et al., 2005]. In contrast, filter predicates in relational settings, where key columns, numbers, or strings are predominantly compared, are much cheaper to evaluate.

In general, early filter execution potentially reduces the set of candidate records dramatically. Therefore, many data flow optimizers(e.g., Pig [Olston et al., 2008], Jaql [Beyer et al., 2011]) contain a heuristic that pushes filter predicates as close as possible to the data sources. In the presence of expensive predicates however, it might be more efficient to first execute other operators that also reduce the number of records

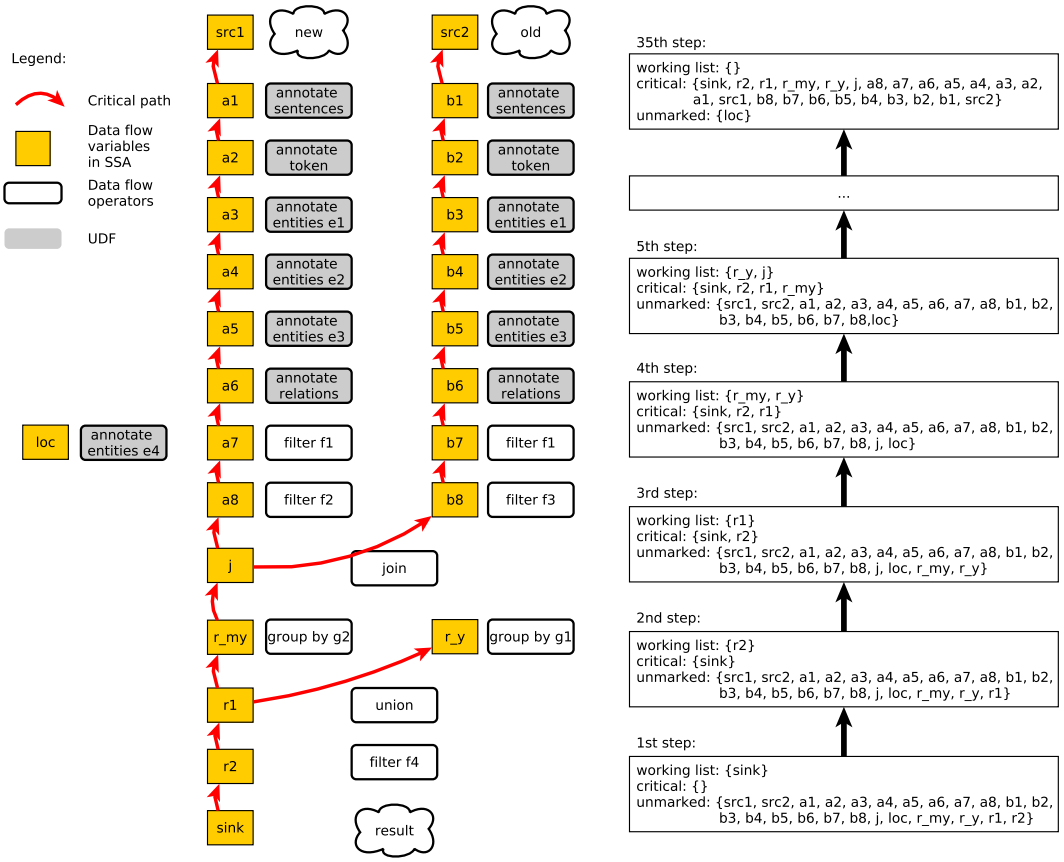


Figure 4.8: Dead code elimination using static single assignment (SSA) and critical path analysis. SSA assignments are contained in yellow boxes beneath each operator and the critical path is marked with red arrows. Data flow edges are not displayed to ease readability.

before evaluating the expensive predicate. Hellerstein proposed a *predicate migration* algorithm, which optimizes the placement of expensive filter operations by assigning a rank to each filter predicate based on selectivity and cost estimates [Hellerstein, 1998]. When ordering filter predicates in ascending rank order in linear data flows without changing the execution order of non-filter operators, Hellerstein showed that this yields optimal execution plans. For non-linear data flows, predicate migration cannot guarantee optimal plans, since in such settings, not only filter ranks need to be considered, but also implicit filter orders need to be respected. For example, a filter that considers attributes from two different data sources cannot be applied before the two data sources themselves are joined. To overcome this limitation, Chaudhuri and Shim [1999] present a different algorithm with complete rank ordering and search space pruning, which is polynomial in the number of user-defined filter predicates.

A different approach for reducing the execution costs of expensive predicates aims at avoiding redundant invocations of expensive predicates or UDFs on duplicate values through caching computed results. A well-known approach for caching the results

of UDFs is memoization, which builds a memory-based hash table storing the UDF's computed values for different input values [Michie, 1968]. Hellerstein and Naughton [1996] combine memoization and sorting in a hybrid caching scheme and show that this approach yields significant improvements over memoization and sorting in most cases.

Respecting expensive predicates is not contained in any data analytics system and to the best of our knowledge, even cost-based data flow optimizers do not address this problem.

4.3.4 Partial aggregation

In contrast to UDFs that process single records one-by-one and which can be executed on arbitrary data partitionings (e.g., `annt`t, `fltr`, `trnsf`), UDFs based on processing groups of records require a certain data partitioning scheme such that all records, which exhibit the same key or grouping attribute, are processed together. To reduce costs for data shipment in situations where a UDF o_1 is executed before a key-based UDF o_2, partial aggregations (so-called *combiners*) can be inserted into a data flow. Combiners pre-compute partial aggregation results locally at the processing site of o_1 directly before sending data to o_2.

Consider Figure 4.9, which shows an excerpt of the join and grouping operators contained in the data flow from Listing 1. We assume that the data flow is executed on four different nodes. In the upper part of the figure, the partial data flow is shown without introducing a combiner and the lower part displays the introduction of combiners. Next to the data flow, the respective record layout of the processed data is shown. Without introducing a combiner, the schema remains unchanged until executing the grouping operator, i.e., many records with many attributes are sent over the network to remote compute nodes. The combiner exhibits the same functionality as the subsequent grouping operation by aggregating joined records by year and projecting out all attributes except of annotated relations. Schema reduction is applied in the combiner before network transfer together with the grouping operation, which significantly reduces the amount of data to be transferred. Finally, the partial results are sent to the appropriate nodes to compute the final grouping.

Combiners can be applied to optimize the computation of averages, correlation, regression, or any other grouping function that is associative and commutative (e.g., *sum*, *min*, or *max*). Combiners are available in all state-of-the-art parallel data analytics systems. Tenzing [Chattopadhyay et al., 2011] also supports hash-based partial aggregations, which need to be specified manually by the user.

Next to partial aggregation before data shipment, placement of aggregation-based operators in data flows is crucial. Push-down or pull-up of aggregations as introduced by Yan and Larson [1994] optimizes the placement of record groupings depending on the selectivity of join operations. Pushing group-bys towards the data sources reduces the number of input records for operations that combine data sets (e.g., join). Pulling group-bys towards the data sinks is deemed beneficial in settings where selective filters are contained in a data flow, which significantly reduce the number of input records for the aggregation operation. Follow-up research [Yan and Larson, 1995] describes eager and lazy aggregation for optimizing settings where the achieved amount of data reduction does not justify to push or pull the entire group-by, but is beneficial only for parts of the input. To perform eager or lazy aggregation, the aggregation function

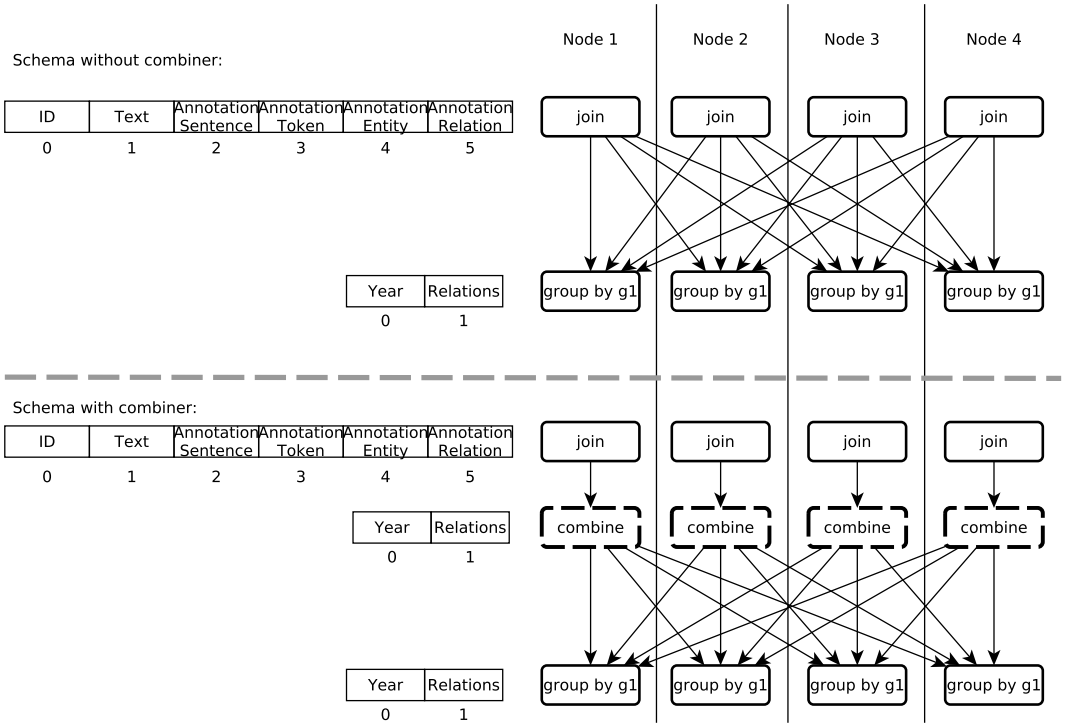


Figure 4.9: Partial data flow executed on four nodes without combiner (top) and with combiner (bottom) to reduce network traffic.

f itself needs to be decomposable, such that when applied to two inputs I_1, I_2 $f(I_1 \cup_a I_2) \equiv f'(f''(I_1), f''(I_2))$ holds. For example, aggregate functions such as *min* or *sum* are decomposable, since $min(I_1 \cup_a I_2)$ is equivalent to $min(min(I_1), min(I_2))$. The decision to perform eager or lazy aggregation should be taken cost based and to our knowledge, this technique has not yet been evaluated or implemented in the context of parallel data analytics systems.

4.3.5 Optimization of communication costs by semi-join reduction and other methods

Network I/O is a main cost factor in data flows with UDFs, since executing such flows on parallel data analytics systems often requires shipping massive amounts of data to remote compute nodes, where expensive computations by the UDFs are performed. In fact, Sarma et al. [2013] found that communication and data shipment costs may dominate the total execution costs of data flows in many cases. Transferring data over the network requires broadcasting, re-partitioning, and shuffling of the data, which greatly impacts the overall performance of data analytics systems. Thus, a main objective during data flow optimization is to reduce communication and data shipment costs as much as possible.

In the context of distributed database systems, semi-join reducers were introduced in the context of join optimization with the goal to identify matching records before

an actual join operator is executed. By inserting semi-join reducers into a data flow, an optimizer can prevent shipping records over the network, which are not part of the final join result. Consider Figure 4.10, which shows a distributed setting with three compute nodes, where a UDF $\bowtie_3 (A,B,C)$ for performing a 3-way join on the data sets A,B,C shall be executed. A is a mid-sized data set with many attributes stored on node 1, B is a large data set with few attributes stored on node 2, and C is a small data set with a medium number of attributes for each record stored on node 3. The computation of the 3-way join can be performed on any of these nodes at different costs. A rule of thumb in such scenarios is to ship the smaller data sets to the nodes where the larger data sets reside to reduce communication costs. Since data set B is the largest data set in our example, a cost-based optimizer decides that all computation should be carried out at node 2, where B is stored and A and C need to be shipped to node 2. Introducing semi-joins at nodes 1 and 3 now attempts to reduce communication costs by first identifying records that qualify for the join condition. Therefore, the data flow optimizer decides to introduce a projection and unification operation on the join attribute a of data set B at node 2 and send this intermediate data set to nodes 1 and 3. At nodes 1 and 3, semi-joins of the form $A \ltimes B$ and $B \rtimes C$, respectively, are executed to identify records that contribute to the final result set of the 3-way join. These records are now sent to node 2, where the final join result is determined.

Instead of sending entire data sets or join attributes over the network, bit vectors [Chan and Ioannidis, 1998; Valduriez, 1987] or Bloom filters [Bloom, 1970] can be applied to exclude irrelevant records without actually evaluating the join predicate. To accomplish this, a bit vector v of size n is created. Each value of the join attribute a of data set B is transformed into a new value in the interval $[1,\ldots,n]$ by using an appropriate hash function and the corresponding bits in a bit vector v are set accordingly. The bit vector v and the hash function are sent to the remote nodes 1 and 3 to identify potential join partners, which are ultimately sent to node 2 for join processing. The size of v shipped to nodes 1 and 3 is significantly smaller compared to performing a projection on the join key column on node 2 and sending the entire column to the remote nodes as employed in semi-join reducers. On the other hand, the size of the set of join candidates shipped by bit vector filtering may be significantly larger due to hash key collisions, since the benefits of filtering highly depend on the choice of a proper hash function and the size of the bit vector. Bit vector filtering is not only valuable for 2-way or multi-way joins, but also for UDFs that involve two phases, such as custom intersections, groupings, etc. In the first phase, such operators can populate a bit vector to remote nodes to skip records in the second phase [Miner and Shook, 2012].

Semi join reducers were first introduced by Bernstein et al. [1981] and later improved by Apers et al. [1983] to reduce data shipment costs in distributed database queries. Although a study showed that this technique was beneficial in distributed database systems in the 80's only for some types of queries [Bernstein and Goodman, 1981], it has been re-discovered in the 1990's and 2000's to optimize different types of queries, for example, star joins or top-n queries [Stocker et al., 2001]. In parallel data analytics systems, where huge data sets are shipped between nodes, this technique is also promising to reduce costs of data transfer, particularly, when n-way joins are involved. To the best of our knowledge, semi-join reducers are currently not contained in any data analytics system's optimizer, but can be added manually by the developer. There are plans to

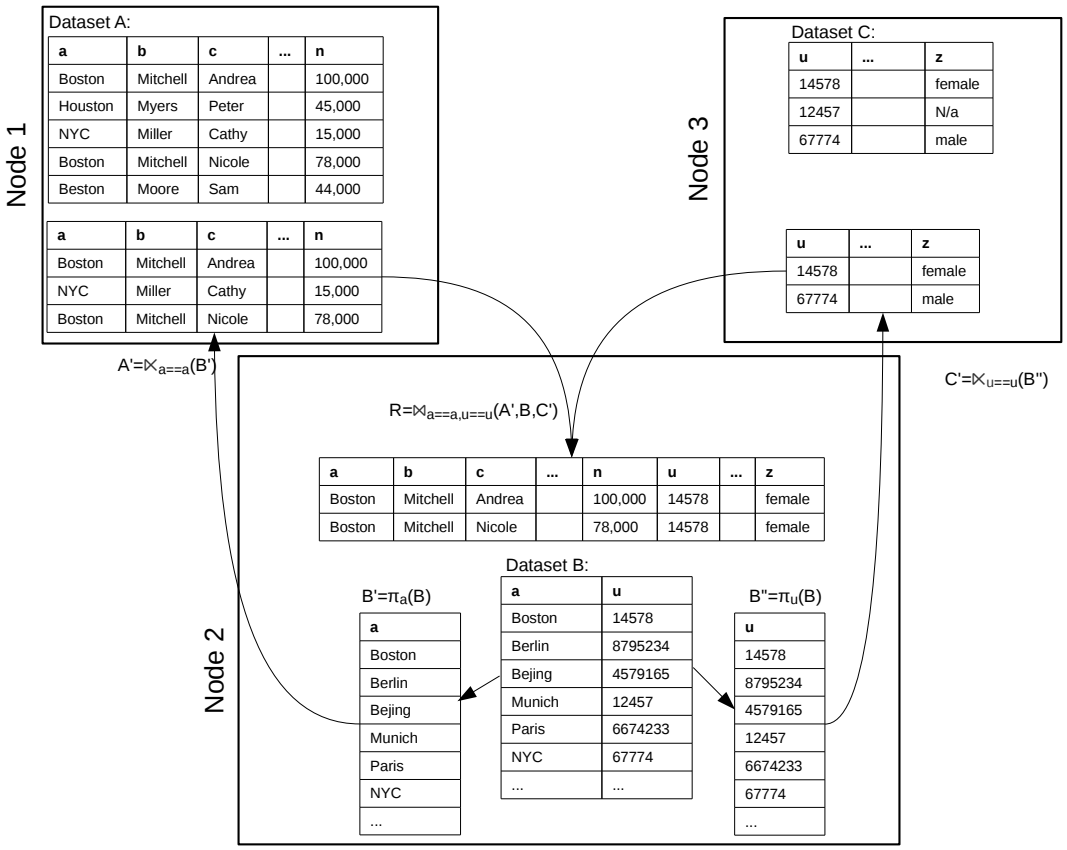


Figure 4.10: Semi-join reduction of user-defined 3-way join operator to decrease network traffic.

integrate semi-join reducers into the Calcite optimizer for Hive with promising initial results, yet, integration has not been finished as of September 2016[17].

Apart from introducing semi-join reducers, the data shipment strategy itself is an important part for adjustment during data flow optimization. Intuitively, when two or more data sets or partitions shall be combined in a distributed setting, two data shipment strategies are deemed beneficial. These strategies are (1) *ship as a whole*, which ships entire data sets to remote nodes, and (2) *fetch as needed*, where compute nodes fetch records as needed. Clearly, both approaches have severe disadvantages, i.e., the first results in high volumes of shipped data and the second yields a vast amount of messages that are exchanged between the compute nodes. Different strategies have been proposed to overcome this problem, namely the introduction of on the fly indexing, data compression, and partition pulling. Another option for optimizing network transfer and data shuffling costs is to pull and replicate entire partitions on certain compute nodes if they are sufficiently small [Graefe, 2009]. In the same way, Alexandrov et al. [2015] propose to re-use existing partitions by first computing sets of interesting partitionings for a given data flow and enforcing such partitioning in early stages of the data

[17] `http://issues.apache.org/jira/browse/CALCITE-468`, last accessed: 2016-10-31.

flow execution. In addition, data analytics systems, such as Flink, Spark, or Cloudera, commonly apply techniques for data compression before shipment across the network.

4.3.6 Choice of operator implementation

Once a logical plan for a data flow is fixed, concrete algorithms and execution strategies for each operator need to be chosen. Choosing appropriate operator implementations is as important as the previous optimization techniques, since data properties such as distribution, sortedness, or cardinality impact operator and data flow performance dramatically. Choosing join algorithms adaptively is discussed in [Blanas et al., 2010]. For UDFs however, choosing appropriate alternatives based on data properties becomes even more difficult because of the lack of appropriate statistics and often, only few different UDF implementations are available to a data analytics system although UDF variants exist. In parallel data analytics systems, annotations of different physical properties, such as input sizes, the ratio between input and output sizes and key cardinalities, and the number of produced records per UDF function call are used by Battré et al. [2010] and Alexandrov et al. [2014] to decide upon a parallelization strategy and to choose specific join operator implementations for data flow execution. The optimizer for Pig contains specific mechanisms to choose appropriate join and aggregation algorithms for skewed data distributions [Gates et al., 2013].

4.4 Data flow languages and optimization in Map/Reduce-style systems

In the past decade, many data flow languages with implementations for different parallel data analytics systems have been proposed. These languages aim at simplifying the writing of data analytics programs and to enable automatic optimization and parallelization of such flows by appropriate compilers and optimizers. In the following, we summarize some of the proposed languages and their key characteristics in terms of type, data model, intermediate representation, targeted Map/Reduce stack, and available optimization techniques regarding UDFs. We compare general language properties in Table 4.1 and technical properties in Table 4.2.

Dremel [Melnik et al., 2010], also known as BigQuery, is a language designed at Google to support interactive ad hoc analyses of very large read-only data sets. BigQuery provides the core set of Dremel language features and is available to external developers whereas Dremel is only available within the company. Next to relational queries, the language supports inter- and intra-record aggregations, top-k queries, and UDFs. Recently, an open-source implementation of Dremel was made available in the Apache Drill [Hausenblas and Nadeau, 2013] project, which compiles both to BigTable and Hadoop to enable execution on large compute clusters. Drill is extensible with UDFs and extensions of Dremel to support tree-structured data have also been proposed [Afrati et al., 2014]. Drill applies a compiler pipeline that employs byte-code analysis and rewriting to enable automatic code optimization. Optimization of aggregation and join queries has to be carried out manually by the developer, UDFs are not optimized.

Tenzing [Chattopadhyay et al., 2011] is another language developed by Google intended to support SQL and relational OLAP applications using batch processing on Map/Reduce infrastructures. A cost-based optimizer is available to optimize aggregations and joins, for which different join algorithms are available. As Dremel, Tenzing is extensible with UDFs, which are not optimized.

PigLatin [Olston et al., 2008] was originally developed at Yahoo research and is a SQL-like declarative scripting language to support analytical queries on top of Hadoop. It is extensible with UDFs and supports relational and arithmetic operators. Though originally developed for the Hadoop stack, plans for compilation to other systems, such as Tez, Spark, or Storm are under development[18]. Optimization is carried out rule-based in a limited, database-style form. Join optimization is carried out manually and techniques to optimize processing of skewed data are available. UDFs are not optimized.

JaQL [Beyer et al., 2011] is a functional scripting language for processing semi-structured data on top of Hadoop, where operators are expressed as functions and data flow programs are compositions of functions. Optimization is carried out rule-based on the logical level and includes variable and function in-lining, filter push-down and optimization of field access. JaQL is extensible with UDFs and user-defined aggregations, which are not optimized.

Hive [Thusoo et al., 2009] is a data warehousing system on top of Hadoop and Spark, which provides HiveQL, a SQL-like query language. It contains a rule-based optimizer capable of optimizing joins and aggregations, predicate push-downs, data pruning, and dynamic partition pruning. Optionally, the optimizer can be replaced by a cost-based optimizer built on top of Apache Calcite[19] that uses cardinality estimates to generate efficient execution strategies.

Scope [Chaiken et al., 2008; Zhou et al., 2012] is one of the earliest declarative scripting languages proposed for large-scale data processing of structured records similar to relational databases. It is extensible with UDFs and contains a limited set of SQL-style operators, which are optimized cost-based by rewriting sub-expressions using the Cascades framework [Graefe, 1995]. Later, an enhanced optimizer has been included that performs code analysis to optimize imperative UDFs using techniques such as column reduction, early filtering, and optimization of data shuffling [Guo et al., 2012]. Recently, Microsoft released a novel query language named U-SQL[20], which provides SQL operations and UDF functionality for the storage and analysis system Azure. Optimization and execution in this system is carried out based on the techniques developed for Scope.

DryadLINQ [Yu et al., 2008; Isard and Yu, 2009] enables data-parallel programming in the .NET environment by extending the declarative-imperative language and programming model LINQ [Meijer et al., 2006]. LINQ has a SQL-like syntax, which is enriched with lambda expressions and anonymous types. Optimization is carried out rule-based and based on annotations of algebraic operator properties provided by the developer. The DryadLINQ project has been stopped in 2010 in favor of Hadoop and Spark, however, an academic version providing source-code is still available.

[18] Information is taken from Pig specification proposals available at `https://cwiki.apache.org/confluence/display/PIG/Pig+on+Spark` for Spark, `https://cwiki.apache.org/confluence/display/PIG/Pig+on+Storm+Proposal` for Storm, and `https://cwiki.apache.org/confluence/display/PIG/Pig+on+Tez` for Tez. All URLs were last accessed on 2016-12-12.

[19] `https://calcite.apache.org/`, last accessed: 2016-10-31.

[20] `http://usql.io/`, last accessed: 2016-10-31.

	Language type	Application area	Target system	Extensibility	Open source
Dremel/Drill	1	a,c	★,♦	◊,□	✓
Tenzing	1	c,d	▲	◊	–
PigLatin	1,2	a,c	★,♣	◊	✓
JaQL	1,2	a,b	★	◊,△	✓
HiveQL	1	c,d	★,♣	◊	✓
Scope/U-SQL	1	b,c	♦	◊,□	–
DryadLinq	1,2	a,c	♦	◊	✓
Meteor/Sopremo	1,2	a,b,c	♠*	◊,□	✓
AsterixQL	1	a,b	■	◊	✓
Spark SQL	1	a,b,c	♣	◊	✓
Sawzall	3	b	▲	–	✓
Impala	1	c	★	◊,△	✓
Jet	1	a,b	★,♣	□	–
Emma	1	a,b	♠	□	–

Legend

Language type	1: declarative, 2: scripting, 3: procedural
Application area	a: general purpose, b: domain-specific, c: relational, d: OLAP
Target system	★: Hadoop, ♦: BigTable, ♣: Spark, ♠: Flink, ♠∗: Stratosphere, ■: Hyracks, ▲: MapReduce (Google)
Extensibility	◊: UDF, □: domain-specific languages, △: user-defined aggregation

Table 4.1: Overview of data flow languages for parallel data analytics systems. Part 1: general language properties.

Meteor [Heise et al., 2012] is a declarative and extensible data flow language for Stratosphere, the research branch of Apache Flink (cf. Chapters 2 and 3). Next to general-purpose operators, it contains many domain-specific operators for information extraction, data cleansing, and web data extraction. UDFs are implemented as first-class algebraic operators, which are provided in self-contained libraries consisting of the operators implementations, their syntax, and optional semantic annotations, which allows an optimizer to access and exploit this information for logical data flow rewriting at compile time [Rheinländer et al., 2015]. Optimization is carried out in a hybrid way, where manual annotation is combined with operator and static code analysis to infer rewrite options (cf. Chapter 5).

AsterixQL (AQL) [Alsubaiee et al., 2014] is a declarative language for AsterixDB, which provides native support for analyzing nested data by using FLWOR statements originally introduced in XQuery. AQL is shipped with many general-purpose and domain-specific operators, for example, to execute similarity-based or range-based queries. Optimization is carried out using algebraic rewrite rules in the Algebrix framework and can be enhanced by manually providing operator annotations. Recently, a connector interface between AsterixDB and Spark has been demonstrated, allowing users to submit AQL queries through Spark to AsterixDB to produce intermediate result partitions, which are processed by Spark for advanced analytics [Alkowaileet et al., 2016].

	Data type	Schema definition	Intermediate format	Optimization of UDFs
Dremel/Drill	■,▲,♣	◊	b	–
Tenzing	■	♦	a,c	–
PigLatin	■,▲,♣	◊	c	–
JaQL	■,▲,♣	◊	c	–
HiveQL	■	♦	a,c	–
Scope/U-SQL	■,♣	♦	a	✓[2]
DryadLinq	■,▲,♣	♦	a,c	✓[2]
Meteor/Sopremo	■,▲,♣	◊	b	✓[1,2]
AsterixQL	■,▲	◊	b,c	✓[2]
Spark SQL	■,▲,♣	◊	a,c	✓[2]
Sawzall	■	♦	c	–
Impala	■	♦	a,c	–
Jet	■,▲,♣	◊	a	✓[1]
Emma	■,▲,♣	◊	a	✓[1]

Legend

Data type	■: structured, ▲: semi-structured, ♣: unstructured
Schema definition	♦: required, ◊: optional
Intermediate format	a: annotated parse tree, b: algebraic data flow, c: plain parse tree
Optimization of UDFs	1: code analysis, 2: semantic annotations

Table 4.2: Overview of data flow languages for parallel data analytics systems. Part 2: Technical properties and optimization.

Spark SQL [Armbrust et al., 2015] is the successor of Shark [Xin et al., 2013] and provides a SQL interface to Spark. Next to relational operators, libraries for domain-specific applications such as streaming, graph processing, and machine learning are available. Spark SQL queries are translated and optimized using Catalyst, an extensible optimizer that performs 2-phase optimization on abstract syntax trees similar to optimization in database systems. Optimization includes many different techniques, however, the optimization of UDFs is not natively supported, but can be added manually through additional rewrite rules.

Sawzall [Pike et al., 2005] is a procedural, domain-specific language developed for the Google Map/Reduce stack for analyzing log records and was made available as open-source in 2010. However, since the underlying runtime system is not publicly available, it can only be applied for analyzing small to mid-sized data sets. An open-source re-implementation of the Sawzall compiler and runtime for the Hadoop stack is available under the name Sizzle[21], an optimizer is not included.

Impala [Kornacker et al., 2015] is an Apache Incubator project initiated by Cloudera that enables real-time SQL queries on top of the Hadoop stack. To enable real-time processing, the Map/ Reduce execution environment of Hadoop is replaced by a parallel and distributed query processor similar to a parallel DBMS. Optimization is carried out in a two-staged approach: a query is translated and locally optimized, subsequently translated into a parallel plan, which is physically optimized and executed.

[21] `http://sizzlelanguage.blogspot.de/`, last accessed: 2016-10-31.

Deep language embedding of domain-specific languages into host languages not only allows for a high-level, declarative task description together with data parallelism transparency but also allows in principle for holistic optimization by means of compiler optimizations and advanced relational and domain-specific optimizations. Jet [Ackermann et al., 2012] is a framework for deeply embedding domain-specific languages for large-scale analytics into Scala on top of Hadoop and Spark. It combines optimization techniques from compilers (e.g. loop fusion, dead code elimination) with mechanisms for projection insertion and operator fusion. Emma [Alexandrov et al., 2015] is also deeply embedded in Scala and uses monad comprehensions in a layered intermediate representation together with a complex, multi-staged compiler-optimizer pipeline to generate efficient code for Flink. Optimization of UDFs is not addressed specifically, yet, UDFs are optimized (as all other functions) during code compilation through the Scala compiler.

4.5 Summary

In this chapter, we surveyed practical techniques for optimizing data flows with UDFs, which are applied at different stages of the optimization process in parallel data analytics systems. Some of the discussed techniques are already available in concrete systems, although comprehensive optimization of UDFs and non-relational operators remains an open challenge. In the next chapter, we present SOFA, a semantics-aware and extensible logical optimizer for data flows with UDFs, which builds upon a semantic analysis of UDF properties for comprehensive data flow optimization.

5 Extensible and semantics-aware optimization of data flows with UDFs

As discussed in Chapter 4, a variety of data flow languages has been proposed that aim at (a) making the definition of complex analytical tasks easier and at (b) allowing flexible deployment of data flows on diverse hardware infrastructures, especially on compute clusters or compute clouds [Sakr et al., 2011], and at (c) supporting domain-specific analysis tasks through the definition of UDFs. Research has shown that a proper optimization of such data flows can improve the execution times by orders of magnitude [Cafarella and Ré, 2010; Hueske et al., 2012; Wu et al., 2011]. However, most optimizers for parallel data analytics systems focus on relational operators, because their semantics in terms of optimization is well understood. In contrast, UDFs can exhibit all sorts of behavior, which are difficult to describe in an abstract, optimizer-enabling manner. As we have seen in the previous chapter, many optimizers for data flow languages treat UDFs essentially as black boxes and disregard them during the optimization.

In this chapter, we address the three most prevailing challenges in semantics-aware optimization of UDFs in parallel data analytics systems, namely

1. defining the most important UDF properties, since properly optimizing UDFs requires properties beyond the classical relational ones,
2. defining appropriate rewrite rules, since novel properties require novel ways of transforming data flows, and
3. finding the right set of properties for describing a given UDF.

We contribute to all three challenges in the following ways: We present *SOFA*, a semantics-aware optimizer for data flows with UDFs. Compared to previous work, SOFA features a richer, yet concise set of general operator properties for automatic and manual UDF annotation. Using these properties and rewrite rules, SOFA is capable of finding a much larger and a more efficient set of semantically equivalent logical plans for a given data flow compared to other systems. Given a concrete data flow, both automatically detected and manually created annotations are evaluated by a cost-based optimizer, which uses a concise set of rewrite templates to infer semantically equivalent plans.

A major obstacle to the optimization of data flows with UDFs is the diversity of the contained UDFs. Our optimizer is developed in Stratosphere, which provides a rich set of custom, domain-specific UDFs together next to relational operators, each implementing one or more of basal second-order functions map, reduce, match, co-group, or cross. As explained in Chapter 2 and Chapter 3, available packages already now contain 51 operators, 35 of which are non-relational, with in total over 150 operator instantiations. Defining semantic properties for each of these operators and rewrite rules for

each possible pair of operators would result in an unacceptable burden to the designer and would considerably limit extensibility and maintainability, since every new operator in principle would have to be analyzed with respect to every existing operator to specify possible rewritings. SOFA solves this problem by means of an extensible taxonomy of operators, operator properties, and rewrite templates called *Presto*. SOFA uses the information encoded in Presto to reason about relationships between operators during plan optimization. Specifically, it leverages subsumption relationships between operators to derive reorderings not explicitly modelled. Presto considerably eases extensions, as novel operators can be hooked into the system by specifying a single subsumption relationship to an existing operator exhibiting the same behavior with respect to optimization; these new operators are immediately optimized in the same manner as their parent. If desired and appropriate, more rewrite rules and operator properties describing the new operator may be introduced later in a pay-as-you-go manner [Roth and Schwarz, 1997].

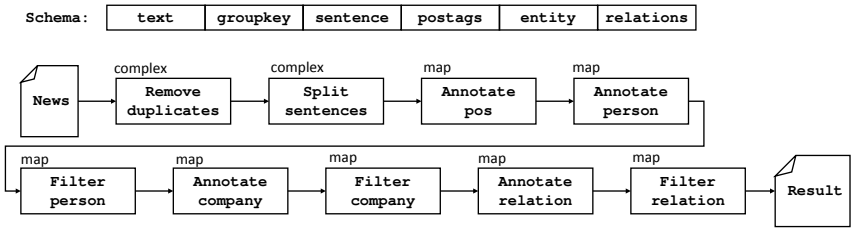
In summary, this chapter presents the following contributions:

1. We identify a small yet powerful set of UDF properties influencing important aspects of data flow optimization in parallel data analytics systems.
2. We show how these properties can be arranged in a concise taxonomy to ease UDF annotation, to enable automatic property inference, and to enhance extensibility of data flow languages.
3. We present a novel optimization algorithm that is capable of rewriting DAG-shaped data flows given proper operator annotations.
4. We evaluate our approach using a diverse set of data flows across different domains. We show that SOFA subsumes existing data flow optimizers in the sense that it enumerates a larger plan space and it finds more efficient plans with factors of up to six.
5. Our experiments show that optimization as carried out with SOFA is even more beneficial when working on very large input data.

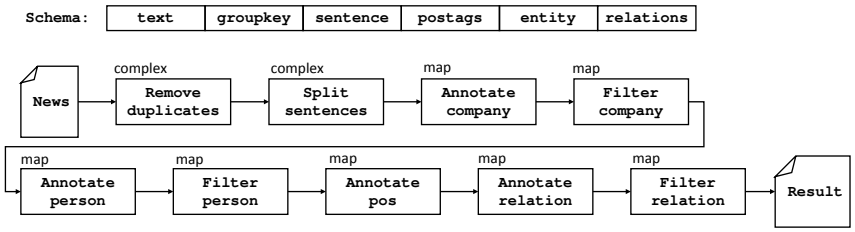
Running example

We use the following example to explain the principles of SOFA throughout this chapter: A large set of news articles shall be analyzed to identify persons, companies, and associations of persons to companies. We assume the articles stem from a web crawl and have already been stripped of HTML tags, advertisements, etc.; still the set contains many duplicate articles, as different news articles are often copied from reports prepared by a news agency.

An exemplary data flow for this task is shown in Figure 5.1(a). The first operator removes duplicates by first computing a grouping key followed by an analysis of each group for similar documents, such that detected duplicates are filtered out per group. Next, a series of operators performs linguistic analysis (sentence splitting and part-of-speech tagging), entity recognition (persons and companies), and relation identification (persons $\leftrightarrow$ companies). After each annotation operator, filter operators remove texts



(a) Initial data flow



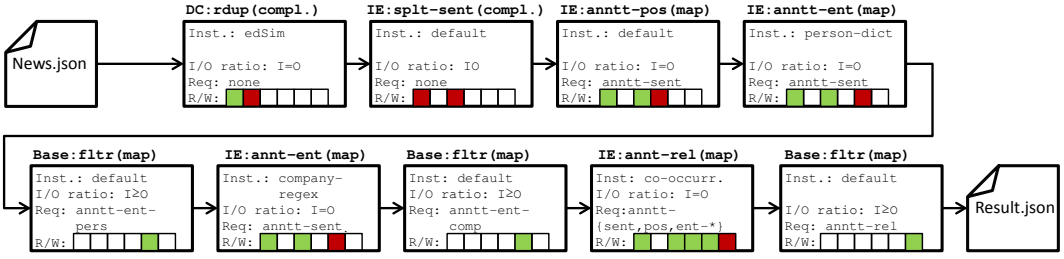
(b) Reordered data flow based on operator semantics

Figure 5.1: High-level data flow for employee relationship analysis.

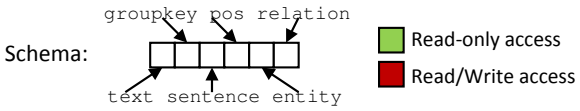
with no person, no company, or no person-company relation, respectively. As displayed in Figure 5.1(a), the data flow is composed of nine operators: seven elementary, and two complex operators (first and second from left). If UDFs are treated as black boxes, this data flow cannot be reordered. But when provisioned with proper information, such as data dependencies or operator commutativities, an optimizer has multiple options for reordering. For example, the part-of-speech tagger can be pushed multiple steps toward the end of the data flow, as annotations produced by this operator are necessary for relationship annotation only. Moreover, the entity annotation operators are commutative, as they independently add annotations to the text, but never delete existing ones. Thus, both annotation operators can be reordered for early filtering. Figure 5.1(b) displays an equivalent data flow with prospectively smaller costs as the most selective filters are executed as early as possible and expensive predicates are moved to the end of the data flow as much as possible. As we will see in Sections 5.1 and 5.4, existing data flow optimizers cannot infer this plan.

The algebraic plan of the logical data flow for our running example is shown in Figure 5.2(a) together with properties and inferred schema information (cf. Figure 5.2(b)), which are used for optimization. Figure 5.2(c) exemplifies that a complex operator may exhibit different properties than its elementary components: The complex operator `splt-sent` has different read/write set annotations and different I/O ratios than its elementary components.

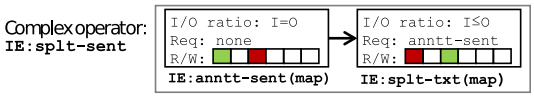
The remainder of this chapter is structured as follows: Section 5.1 gives an overview of our approach for data flow optimization. We also demonstrate how this plan can be reordered substantially by exploiting information on operator semantics. Details on Presto and SOFA are explained in Sections 5.2 and 5.3, and we evaluate our approach in Section 5.4. The user interface of SOFA is briefly described in Section 5.5 and we summarize this chapter in Section 5.6.



(a) Algebraic data flow



(b) Global schema



(c) Resolution of complex operator `splt-txt`

Figure 5.2: Algebraic data flow for the running example. Subfigure (a) displays concrete operator instantiations together with properties relevant for optimization. Colored boxes indicate read/write access on record attributes, which are part of the global schema shown in subfigure (b). Subfigure (c) shows the resolution of the complex `splt-sent` operator (second operator in (a)) into its components `anntt-sent` and `splt-txt`.

This chapter has been previously published as [Rheinländer et al., 2015] and [Rheinländer et al., 2014].

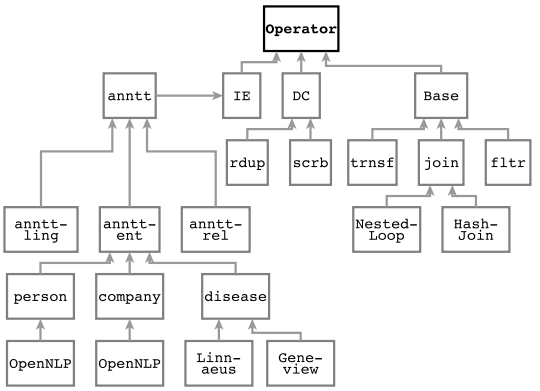
5.1 Semantics-aware data flow optimization by example

SOFA is an optimizer for rewriting data flows with UDFs into semantically equivalent data flows whose expected efficiency is higher according to a cost model and it is capable of introducing, removing, and reordering operators. Complex operators are optimized both as a whole and after expansion. In the following, we focus on operator reordering and complex operator resolution, which are the most intricate and most effective optimization techniques.

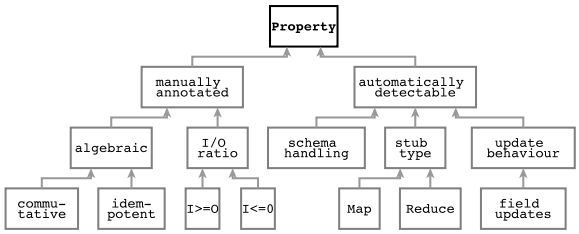
Rewriting depends on a set of rewrite rules, each defining valid manipulations of data flow sub-plans, such as a reordering of two filter operations [Graefe, 1994]. The novelty of SOFA lies in its flexible and extensible treatment of Map/Reduce-style UDFs beyond the capabilities of existing approaches. In this section, we highlight the advantages of SOFA by means of our running example.

Existing approaches for data flow optimization enable reorderings by using either manually defined rewrite rules for relational operators [Olston et al., 2008] or by performing some kind of code analysis [Cafarella and Ré, 2010; Hueske et al., 2012]. The approach of Hueske et al. [2012] probably is the most general, as it automatically derives data flow reorderings based on read/write set analysis of individual UDFs. In particular, the order of two subsequent tuple-at-a-time operators may safely be switched if they have no read/write or write/write conflicts on any attribute (cf. Section 4.2 for details). The data flow shown in Figure 5.2 allows only one such beneficial reordering: The `anntt-pos` operator, which annotates part-of-speech tags and stores them in the fourth attribute (first row, third from right), can be pushed before the `anntt-rel` operator (second row, second from right), because part-of-speech annotations are accessed only during relation annotation. This reordering most likely saves costs, because the different `fltr` operators are now executed before `anntt-pos` and thus fewer sentences have to be annotated. Moving `anntt-pos` towards the start of the data flow is not possible, because it reads annotations produced by the complex `splt-sent` operator.

Semantics-aware rewrite rules allow to reorder the data flow in Figure 5.2 more extensively. Consider the two `anntt-ent` operators. Both write into the same attribute (the fifth), which collects all entity information. If the optimizer knows that annotation operators only *add* values and never delete or update existing values, these operators together with their subsequent filter operators may be reordered. The best order very likely is the one that filters the most sentences first; this decision can make use of selectivity and execution time estimates at the operator level (see Section 5.3). Furthermore, the optimizer can decompose complex operators and reorder the components individually. For instance, `splt-sent` consists of an `anntt-sent` operator and a UDF splitting the input text into separate sentences based on the annotations produced by `anntt-sent`. As shown in Figure 5.2(c), the two components of `splt-sent` differ in terms of read and write access on attributes and I/O ratio. Pushing `splt-txt` some steps towards the end of the plan is valid, because all succeeding `anntt` operators perform their analyses sentence-based and all `anntt` operators for entity, relation, and part-of-



(a) Operator taxonomy



(b) Property taxonomy

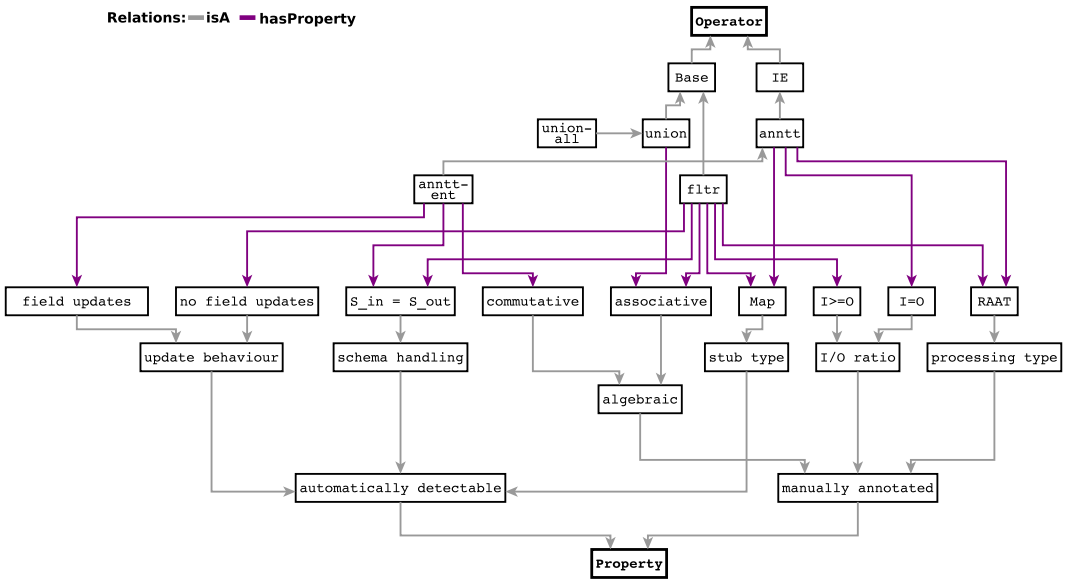
Figure 5.3: Exemplary subgraphs of Presto operator (a) and property (b) taxonomies; root nodes are displayed in bold.

speech annotation read sentence annotations. This reordering is likely beneficial, as `splt-txt` generates multiple output records for each incoming record, depending on the number of annotated sentence boundaries.

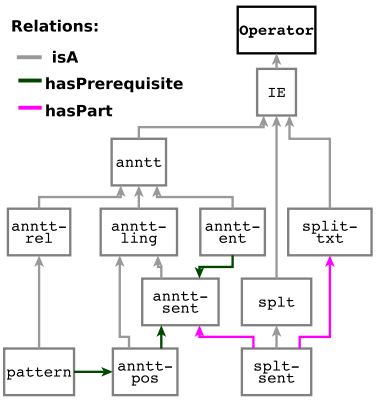
In summary, semantics-aware plan rewriting allows us to pick a plan (with respect to cost estimates) from a larger set of equivalent data flows compared to other existing approaches. For instance, SOFA finds 4,545 distinct plans for the running example, compared to only 512 plans found with the read/write-set analysis of [Hueske et al., 2012] (see Section 5.4 for a detailed comparison).

5.2 The Presto taxonomy for annotating and rewriting UDFs

To enable optimizations such as those shown in the previous section, operators need to be annotated with meta data, for instance to describe selectivity estimates or semantic properties, such as associativities or commutativities. In this section, we introduce Presto, an extensible taxonomy for annotating and rewriting operators. Presto consists of two major components, the operator-property graph for modelling relationships between operators and properties, and a set of rewrite templates for data flow rewriting. When designing Presto, we paid special attention to extensibility by allowing enhance-



(a) hasProperty



(b) hasPrerequisite/hasPart

Figure 5.4: Exemplary subgraphs of relationships hasProperty (a), hasPart (b), and hasPrerequisite (b) between nodes in Presto operator and property taxonomies; root nodes are displayed in bold.

ments to the semantic operator descriptions over time to more and more unleash their optimization potential.

5.2.1 Operator–property graph

The operator-property graph in Presto contains two taxonomies for classifying operators and properties. Both taxonomies are self-contained and model generalization–specialization relationships (*isA*) between operators and properties, respectively. Figures 5.3(a) and 5.3(b) display subgraphs of Presto. For example, the `anntt` operator has two specializations: `anntt-ent` and `anntt-rel` shown in Figure 5.3(a). Each leaf in the operator taxonomy describes a concrete implementation of the parent operator. Concrete implementations of operators can be very diverse: Similar to the relational world where multiple algorithms for the join operator exist, a non-relational operator for removing duplicates can be implemented naïvely in a theta join, by using a multi-pass sorted neighborhood approach, or through other advanced algorithms [Hernández and Stolfo, 1995]. These implementations have different properties, for example, a theta join is an elementary operator which is implemented in a "cross" parallelization function, whereas the sorted neighborhood implementation is a complex operator consisting of multiple elementary operators [Heise, 2015]. Depending on which implementation of the duplicate removal operator is chosen (this can either be defined by the user or the system selects a default implementation), the potential for data flow reordering and the concrete data flow transformations may differ. The design of the operator taxonomy allows us to uniquely identify available operator instantiations, to use subsumption to effectively assign properties and relationships to operators, and to deduce rewrite options. Note that such abstraction-implementation relationships are an established concept in relational optimizers. However, in the relational world the hierarchies are very flat; they become much deeper when dealing with domain-specific UDFs.

As shown in Figure 5.3(b), we distinguish between automatically detectable properties and properties that are annotated by the package developer. The latter comprises algebraic properties (e.g., commutativity, associativity), processing type (record-at-a-time, bag-at-a-time), and the ratio between the number of input and output records. Automatically detectable properties comprise the parallelization function of the operator implementation (e.g., map, reduce), schema information available at compile time, the number of inputs, and the read/write behavior. Note that Presto defines only three manual and four automatically detectable property types; assigning these in an effective and intuitive manner to large sets of UDFs is the core rationale behind Presto.

Relationships connect operators and properties. Each specialization inherits all properties and relationships that are defined for the corresponding generalizations. For instance, the `union-all` operator shown in Figure 5.4(a) is a specialization of the `union` operator and thus inherits the algebraic properties defined for `union`. Complex operators can be characterized with respect to their components using the *hasPart* relation (Figure 5.4(b)). For example, the complex operator `splt-sent` consists of the two elementary components `anntt-sent` and `splt-txt`.

Next to *isA* and *hasPart*, we define a *hasProperty* and a *hasPrerequisite* relation. *HasProperty* is a binary relation between an operator and a property and is used to

characterize operator semantics. For instance, the following properties are attached to `fltr` (Figure 5.4(a)):

- is implemented with a Map function,
- does not modify inside fields,
- input $\geq$ output, and
- is commutative to other `fltr` instantiations.

Precedence constraints between operators are captured with *hasPrerequisite(X,Y)*, which states that operator X must be executed before operator Y. In Figure 5.4(b) it is shown that `anntt-rel` based on linguistic patterns requires part-of-speech and entity annotations to be performed in advance. Since `anntt-ent` itself requires sentence annotation and *hasPrerequisite* is a transitive relation, it is necessary to apply `anntt-sent` before `anntt-rel`.

The `isA` relationship simplifies derivation of novel rewrite options for operators that are initially not well annotated. Suppose, the data scrubbing operator `scrb` from Stratosphere's data cleansing package is initially not equipped with any *hasProperty* relationships. Later, the developer may see that `scrb` is a specialization of the well-annotated `trnsf` operator from the Base package, i.e., both operators perform write operations in attributes of the incoming records. By formally specifying this through an *isA* relationship, `scrb` inherits all properties defined for `trnsf` (not shown in Figure 5.3(a)).

Though the complete Presto graph is too large to show here, it is still rather small and easy to understand: The property taxonomy contains 32 nodes and the operator taxonomy 117 nodes. An overview of all prerequisites (*hasPrerequisite* relationship), operator properties (*hasProperty* relationship), and elementary components of complex operators (*hasPart* relationship) for all IE and WA operator instantiations is listed in tabular form in Appendix 1. Note that new packages mostly extend the operator taxonomy, while the property taxonomy is a fairly stable structure in our experience.

5.2.2 Rewrite templates

We perform data flow rewriting using a set of rules specifying semantically valid reorderings, insertions, or deletions of operators. Because rewrite rules apply to combinations of operators, and because the different independently developed and maintained packages available for Stratosphere already contain more than 70 individual operators, it is practically impossible to define all rewrite rules across the different packages one-by-one. Instead of explicitly formulating each possible order of executing any two operators as done for the concise set of relational operators, we define a concise set of rewrite templates, which consist of rather general operator properties and abstract operators as building blocks. Reasoning along relationships modelled in the Presto taxonomy allows SOFA to automatically instantiate the templates with concrete operators and thus enables us to derive individual rewrite options for concrete operator combinations on the fly. Currently, SOFA requires only 11 rewrite templates, which are expanded to over 150 individual rewrite rules.

Listing 5.1 displays a subset of the available templates in Datalog notation; further rules cover different reorderings based on algebraic properties as well as insertion and removal of operators. The complete set of rewrite templates together with rewriting examples can be found in Appendix 2. The first three templates of Listing 5.1 are static

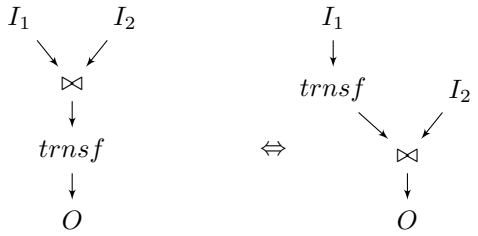
and can be evaluated at package loading time, whereas the last two templates are dynamic and are evaluable at compile time only. The first template covers commutative operators and expresses that two consecutive appearances of operators X annotated as associative in Presto can be safely reordered. Specifically, the goal `reorder(X,X)` evaluates to true if Presto contains a *hasProperty*-relationship of X with the property *associative*. Note that associativity does not necessarily need to be defined directly on X; the rule also applies if some ancestor of X in Presto is marked as associative. This fact is inferred using inheritance rules for reasoning over the Presto graph. The second template (Line 3) enables reordering of operators based on the *isA* relation and states that for any three operator instantiations X,Y,Z, the operators X,Y are reorderable given that X is not a prerequisite of Y, X is a specialization of Z, and Y,Z are reorderable. We include the goal `not hasPrerequisite(Y,X)` in the templates to ensure that operator precedences are respected. The third template (Line 5) enables reorderings of consecutive `anntt` operators X,Y, when X is not a prerequisite of Y.

Listing 5.1: Exemplary rewrite templates.

```
1  reorder(X,X) :- hasProperty(X,'associative'), isA(X,'operator').
2
3  reorder(X,Y) :- not hasPrerequisite(Y,X), isA(X,Z), reorder(Z,Y).
4
5  reorder(X,Y) :- not hasPrerequisite(Y,X), isA(X,'annotate'),isA(Y,'annotate').
6
7  reorder(X,Y) :- hasProperty(X,'single-in'), hasProperty(X,'RAAT'),
8                  hasProperty(Y,'RAAT'), hasProperty(Y,'single-in'),
9                  noReadWriteConflicts(X,Y).
10
11 reorderWithLeft(X,Y) :- hasProperty(X,'dual-input'), hasProperty(Y,'single-input'),
12                         hasProperty(Y,'RAAT'), not contains(readSet(Y),readSet(X)),
13                         not contains(rightInputSchema(X),readSet(X)).
```

Dynamic rewrite templates are partly based on information not available before the data flow is compiled, for example, information on concrete attribute access by operators is available only after posing a Meteor query to the Stratosphere system. Template 4 (Lines 7–9) enables reordering of two single-input record-at-a-time operators if these operators have no read/write conflicts. This single rule essentially covers most optimization options achieved by [Hueske et al., 2012], which shows the power of our approach.

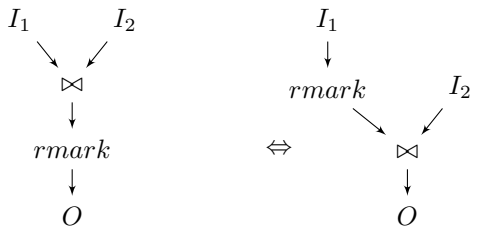
While most rules in Presto are generic and apply to many operator combinations, other rules are more specific. Suppose, we are given a data flow that consists of an equi-join of two data sources I_1,I_2 followed by `trnsf` that transforms only attributes of I_1, which are not part of the join condition. This data flow can be rewritten into an equivalent data flow, which first applies `trnsf` to I_1 and afterwards joins I_1 and I_2 by means of Template 5 (Lines 11–13):



Similar to extending Presto with new operators and properties, package developers can also extend the set of rewrite templates to enable data flow optimization for their concrete application domain. For example, the third template was added by the IE package developer, since it enables reordering `anntt` instantiations, which is not supported by any other Presto template.

5.2.3 Pay-as-you-go annotation of operators

A key feature of SOFA is its extensible design, which significantly reduces the effort for annotating properties of new operators based on the subsumption hierarchy and inheritance mechanisms contained in Presto. By adding an *isA*-relationship for some operators X and Y, X inherits all properties from Y, and thus, X is optimized in the same manner as Y. Similarly, the property taxonomy can be extended with new properties (e.g., requires sorted input, computes aggregate function) and novel rewrite templates using these properties can be added if required for optimizing novel operators. Suppose, the boilerplate detection operator `rmark` from the web analytics package, which detects and removes HTML markup in web pages, is newly integrated into Stratosphere. Initially, this operator would probably not be equipped with any Presto annotations. In this case, the SOFA optimizer can infer only automatically detectable properties, i.e., reordering can be performed only on the basis of read/write-set analysis. Later, the package developer invests some thought and annotates that `rmark` outputs as many records as incoming ($|I| = |O|$). SOFA infers from the set of automatically detectable properties, that `rmark` is a single-input operator implemented with a map parallelization function. Taken these properties together, the last template of Listing 5.1 becomes applicable to `rmark`. A full specification of `rmark` would include the definition of *isA* relationships to other operators. Actually, `rmark` has the same semantics as the `trnsf` operator from the Base package, as it essentially performs a transformation of the input texts. Now all templates valid for `trnsf` become applicable, such as the rule for reordering a `join` and a `trnsf` operator introduced in Section 4.2. Given that `rmark` accesses only attributes present in input I_1 that are not part of the join condition, SOFA can then reorder a data flow containing `rmark` and `join` as follows:



5.3 Optimization Algorithms

SOFA enumerates plan alternatives for a given data flow through data flow transformations based on the available rewrite templates. An overview of the optimization process with SOFA is shown in Figure 5.5 Given a data flow D, SOFA performs two passes of the following three steps. First, D is analyzed for precedence relationships between operators based on rewrite templates and operator properties contained in Presto. This analysis yields a precedence graph, which is used in the plan enumeration phase, to secondly enumerate and thirdly rank valid plan alternatives based on a cost model. Afterwards, the complex operators contained in D are resolved into their elementary components and the three steps are repeated. Finally, the best plan is selected, translated, and physically optimized for parallel execution by the underlying execution engine (see [Alexandrov et al., 2014] for details on this step). Note that the enumeration algorithm of SOFA is not complete as it does not explore any possible combination of complex and elementary operators together in a plan, so we cannot guarantee that SOFA finds the best possible plan. Yet, we will show in the following that SOFA always picks the plan with smallest estimated costs from the search space.

5.3.1 Precedence analysis

Presto models dependencies between operators either explicitly on the basis of the *hasPrecedence* relation or inferred, if the goal `reorder(X,Y)` fails for two operator instantiations X, Y.

Precedence graph construction starts by creating the directed transitive closure D^+ of the given data flow D, which explicitly models all pairwise operator execution orders in D. The algorithm then inspects D^+ to detect and remove edges that are not logically required. It retains all edges incident to a data source or a data sink to prevent reordering of sources and sinks. The goal `reorder(X,Y)` is instantiated with start and end node of each edge (u, v) and the inference mechanism tries to resolve the goal based on the operator properties and rewrite templates stored in Presto. If successful, both nodes are reorderable and the edge (u, v) is removed from the precedence graph. Precedence analysis is a polynomial time algorithm; its complexity is determined by computing the transitive closure in $O(|V|^3)$ using the Floyd-Warshall algorithm and the data complexity of stratified non-recursive Datalog, which we use for reasoning in Presto [Dantsin et al., 2001].

Figure 5.6 shows the final precedence graph for our running example (omitting data sources and sinks for readability). The displayed graph reflects precedences between DC, IE, and Base operators, for example, `rdup` and `anntt-ent-person` are a prerequisite for the `fltr-person` operator, and `anntt-rel` is in a `hasPrerequisite` relation with `anntt-pos` (cf. Figure 5.4(d)). The graph contains edges between `anntt` and `fltr` reflecting that the concrete instantiations of `fltr` have read/write conflicts with their preceding `anntt` operators.

5.3.2 Plan enumeration

Plan enumeration essentially generates different topological orders constrained by the precedence graph, while performing cost-based pruning. In contrast to topological

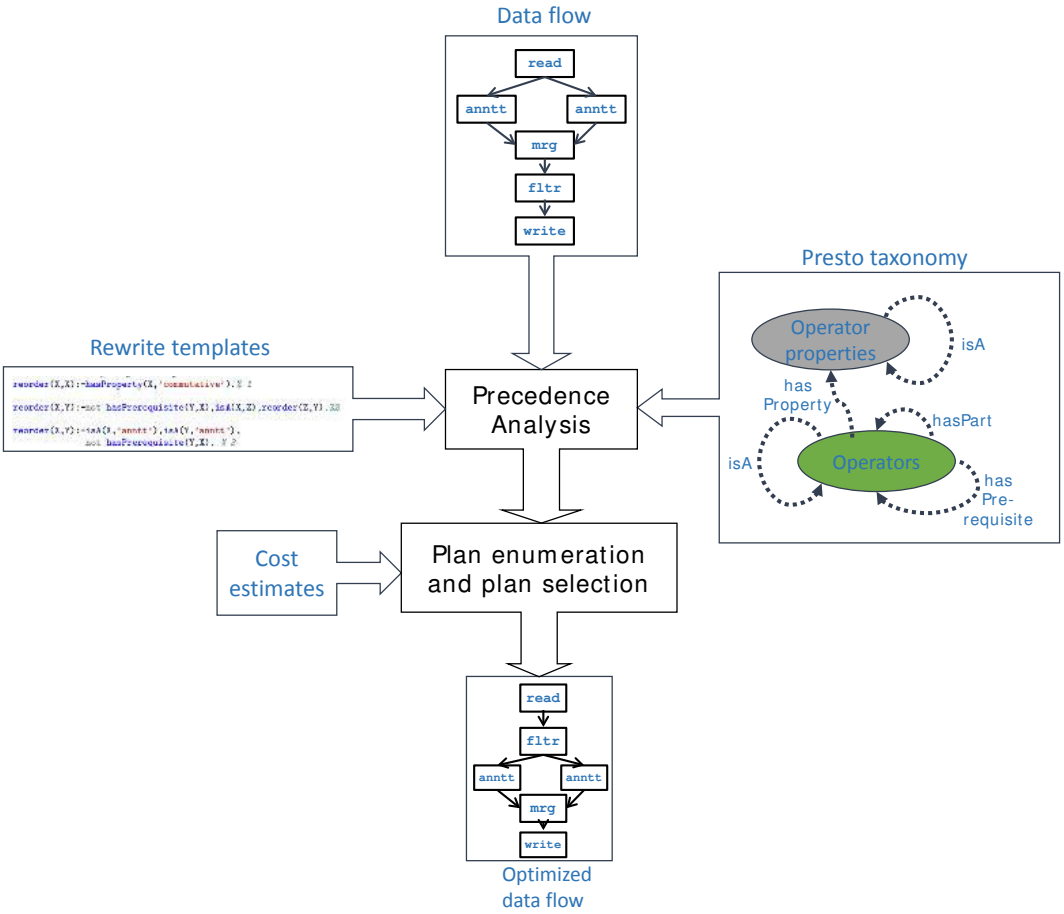


Figure 5.5: Overview of SOFA's data flow optimization process.

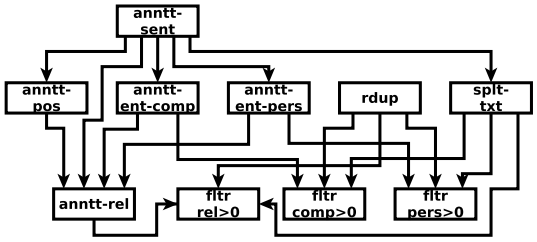


Figure 5.6: Precedence graph for running example with complex operator resolution.

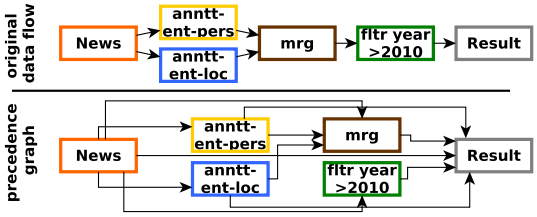


Figure 5.7: DAG-shaped data flow (top) and corresponding precedence graph (bottom) inspired by the running example.

sorting, the outcome are not full orders but DAG-shaped plans. The main idea is to iteratively construct alternative plans from data sinks to sources for a given data flow D by analyzing the corresponding precedence graph for operators that have no outgoing edges. Such operators are not required by any other operator and can therefore be added to the emerging partial plans. If multiple operators have no outgoing edges, the algorithm creates a set of alternative partial plans. The algorithm continues to pursue each alternative, removing the newly added operator from the precedence graph, estimating the costs of the partial plan (see Section 5.4), and pruning costly partial plan alternatives where possible.

We explain its principles using the simplified data flow shown in Figure 5.7 (top). Note that this data flow is DAG-shaped, which poses no problem to SOFA. The data flow performs task-parallel annotation of persons and companies. Annotations are subsequently merged, and the result set is filtered for articles published after 2010. The resulting precedence graph is displayed in Figure 5.7 (bottom). Figure 5.8 shows all stages of enumerating the plan space for our data flow. Columns are alternative partial plans grouped into stages of the algorithm. Boxes correspond to operators with isochromatic frames as defined in Figure 5.7.

The recursive plan enumeration algorithm is displayed in Listing 5.2. It takes as input the original data flow, the corresponding precedence graph, and a partial plan, which initially is empty (Line 1). First, the algorithm selects the set of nodes from the precedence graph that have out-degree 0 (Line 8). These operators are not a prerequisite of any remaining operator and can thus be added to the partial plan without violating precedence constraints. For each of these operators, alternative partial plans are constructed in the following loop (Lines 10–37). In our example, only the data sink can be selected. Once added to the partial plan, the selected node is removed from the precedence graph (Line 11-12). We determine the set *inputNodes* of operators contained in the partial plan having open inputs, i.e., at least one of the input channels of such an operator i is not connected to the output channel of some other operators or a data source preventing a proper functioning of i (Line 13). Since the partial plan was empty before adding the data sink, we cannot insert any edges in the partial plan and therefore, plan enumeration is recursively invoked again (Lines 15–16). Now, `mrg` and `fltr` both have no outgoing edges any more and are therefore added to the set of candidate nodes. Each candidate node is processed individually, added to the partial plan and removed from the precedence graph. This yields in two alternative partial plans, which are both inspected further.

We exemplarily follow the plan with the `mrg` operator. The `mrg` operator is added to the plan and the set of *inputNodes* is divided into required and optional nodes (Lines 18–22). Required nodes are those nodes that have the currently added node as its direct predecessor in the original data flow, optional successors are all other operators contained in *inputNodes*. In our example, the set of required nodes is empty, and the set of optional nodes contains `fltr`. For each required node m, we create an edge (n,m) for the newly added node n, add it to the edge set of our partial plan, estimate the costs of the partial plan, and recursively call the plan enumeration algorithm (Lines 24–29). Each optional node l is processed individually. We iteratively create edges (n,l), estimate the costs of the new partial plan, and again recursively call the plan enumeration algorithm if necessary (Lines 31–36). A recursive invocation of the plan enumeration algorithm terminates either if the precedence graph is empty and an alternative plan has been found (Lines 3–6), or if no alternative plans with smaller costs compared to the initial plan were found (Lines 38–39).

Pruning

The plan enumeration algorithm has exponential worst-case complexity (consider for instance a precedence graph without any edges). We included a simple technique for search space pruning in our algorithm preventing completion of partial plans whose estimated costs are higher than the estimated costs for the current best data flow. Once a cheaper plan was found, we update the costs of the best plan, in a manner similar to accumulated cost pruning in top-down query optimization [Graefe, 1994, 1995]. If no alternative plan with lower estimated costs compared to the best plan could be constructed, we terminate (cf. Listing 5.2, Line 33).

5.3.3 Cost estimation

To estimate costs and result sizes of a data flow, SOFA depends on estimates for key figures of operators, which can either be provided by the developer by adding appropriate annotations to Presto, by sampling from the input data, or by runtime monitoring of previously executed data flows. We estimate the costs of a plan by computing the weighted sum of estimated ship data volume, I/O volume, and CPU usage of the UDFs per call.

Specifically, the costs of an operator o_i are estimated as follows: let c_i be the average CPU usage of o_i per invocation, s_i the estimated startup costs of o_i, and r_i the estimated number of processed input items of o_i. Including startup times of operators is particularly important for complex non-relational UDFs, as many IE and DC operators need a long startup time for instance to load large dictionaries, or to assemble trained models (cf. Section 3.4). Furthermore, let d_i denote the estimated I/O costs of an input item processed by o_i, n_i the estimated shipping costs of an output item produced by o_i, and sel_i the selectivity of o_i. The estimated number of items r_i processed by an operator o_i depends on the number of its preceeding operators h (i.e., an edge $(h,i) \in E(D)$ exists in D) and the selectivities of h and is calculated as follows:

$$r_i = \sum_{(h,i) \in E(D)} r_h * sel_h \tag{5.1}$$

Listing 5.2: Plan enumeration with SOFA.

```
enumAlternatives(Graph precedGraph, Graph plan, Graph partialPlan) {

        if (isEmpty(precedGraph)) {
                addPlanToResultSet(partialPlan);
                return;
        }

        candNodes = getNodesWithOutDegreeZero(precedGraph);

        foreach(Node n in candNodes) {
                addNodeToPartialPlan(n);
                removeNodeAndIncidentEdgesFromPrecedenceGraph(n);
                inputNodes = getNodesWithOpenInputs(partialPlan);

                if (isEmpty(inputNodes))
                        enumAlternatives(precedGraph, plan, partialPlan);

                foreach(Node m in inputNodes){
                        if (inputGraphcontainsEdge(n,m))
                                addNodeToRequiredNodes(m);
                        else addNodeToOptionalNodes(m);
                }

                if(not isEmpty(requiredNodes)) {
                        addEdgesToAllRequiredNodesInPartialPlan(m);

                        if (costs(partialPlan) < costs(originalPlan))
                                enumAlternatives(precedGraph,plan,partialPlan);
                }

                foreach(Node l in optionalNodes) {
                        addEdgeToPartialPlan(n,l);

                        if (costs(partialPlan) < costs(originalPlan))
                                enumAlternatives(precedGraph,plan,partialPlan);
                }
        }
        addPlanToResultSet(plan);
        return;

}
```

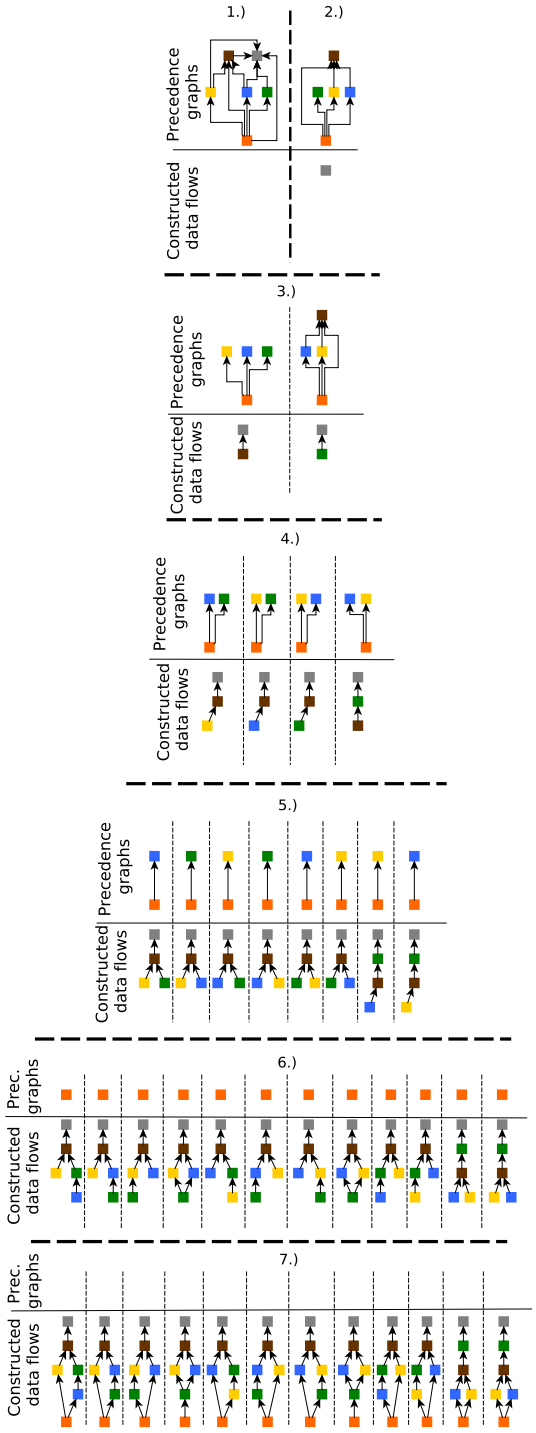


Figure 5.8: Plan enumeration for the DAG-shaped data flow from Figure 5.7. Columns are alternative partial plans grouped into stages of the algorithm. Boxes correspond to operators with isochromatic frames as defined in Figure 5.7.

The costs of an operator o_i are estimated as the weighted sum of the estimated ship data volume, I/O volume, and CPU usage of o_i using the following formula, where u, v, w denote weight constants for each cost component:

$$costs(o_i) = w * (c_i * r_i + s_i) + u * (d_i * r_i) + v * (n_i * r_i * sel_i) \tag{5.2}$$

Note that Formula 5.2 for operator costs can be replaced with custom cost functions, which are added to Presto by the package developers. For example, to accurately estimate the costs for data flows containing IE operators, we also capture the *projectivity* of `anntt` operators, i.e., the average number of annotations produced by an `anntt` instantiation. Consequently, the selectivity of a `fltr` operator, which filters results produced by `anntt` is denoted as $sel(fltr) = r_{i-1} * proj(anntt)$.

Finally, the total costs of a data flow D are estimated as follows:

$$costs(D) = \sum_{i=1}^{n} costs(o_i) \tag{5.3}$$

Note that our cost model optimizes for total computation time, disregarding parallelization in the underlying execution engine. Physical optimization of data placement and shipment between nodes handled downstream by the underlying parallel execution engine. During logical optimization with SOFA, we have no access to the information which concrete shipping strategy will be chosen. Therefore, we assume in our cost model that if two operators o_1, o_2 are implemented in a map function and there is a data flow from o_1 to o_2, the data is not transferred over the network. In all other cases, we assume that the data is shipped over the network. If some dual-input operator o_3 receives inputs from operators o_4 and o_5, we compare the estimated size of the outputs of o_4 and o_5 and assume that the smaller output is transferred over the network. However, we see in Section 5.4 that this approach already allows us to correctly rank enumerated plan alternatives in many cases.

5.4 Evaluation

We evaluated SOFA on a 28-node cluster, each equipped with a 6-core Intel Xeon E5 processor, 24 GB RAM, and 1TB HDD using Stratosphere 0.2.1.

Queries

We implemented, optimized, and executed seven Meteor queries originating from different application domains. These queries are translated into logical Sopremo data flows and handed to SOFA for logical optimization. The concrete Meteor scripts we used for the evaluation are listed in Appendix 3. **Q1** adopts the data flow described in our running example for relationship extraction from biomedical literature using UDFs from the IE and DC packages. **Q2** performs topic detection by computing term frequencies in a corpus grouped by year. The query first splits the input data into sentences, reduces terms to their stem, removes stop words, splits the text into tokens, and aggregates the token counts by year. **Q3** extracts NASDAQ-listed companies that went bankrupt between 2010 and 2012 from a subset of Wikipedia. This query takes article

versions from two different points in time, annotates company names in both sets and applies different `fltr` operators and a `join` to accomplish the task. **Q4** corresponds to the data flow shown in Figure 5.7 and performs task-parallel annotation of person and location names. **Q5** analyzes DBpedia to retrieve politicians named 'Bush' and their corresponding parties using a mixture of DC and base operators. **Q6** is a relational query inspired by the TPC-H query 15. It filters the lineitem table for a time range, joins it with the supplier table, groups the result by join key, and aggregates the total revenue to compute the final result. **Q7** uses two complex IE operators to split incoming texts into sentences and to extract person names.

Data sets

We evaluated Q1 on a set of 10 million randomly selected citations from Medline, Q2 was evaluated on a set of 100,000 full-text articles from the English Wikipedia initially published between 2008 and 2012, Q3 was evaluated on two sets of English Wikipedia articles of 50,000 articles each, one set from 2010 and one set from 2012, Q4 and Q7 on a set of 100,000 full-text articles from the English Wikipedia downloaded in 2012, Q5 on the full DBpedia data set v. 3.8, and Q6 was evaluated on a 100GB relational data set generated using the TPC-H data generator. For each experiment, we report the average of three runs. Estimates on operator selectivities, projectivities, startup costs, and average execution times per input item were derived from 5% random samples of each data set.

Competitors

Although data flow optimization is important in current research, surprisingly few systems actually optimize the data flow at the logical level as we do (cf. Chapter 4). Thus, detecting appropriate competitors is difficult, because optimizers are commonly deeply coupled to a particular system. We reimplemented the ideas of three current data flow optimizers, namely techniques presented by Hueske et al. [2012],Olston et al. [2008], and Simitsis et al. [2005]. We compare the number of plan alternatives found and the achieved runtime improvements. For each method, we disabled rules and information on operator properties stored in Presto and replaced them with the appropriate rewrite rules described in [Hueske et al., 2012; Olston et al., 2008; Simitsis et al., 2005]. For the method of Olston et al. [2008], we referred to the online documentation of rewrite rules for Apache Pig, version 0.11.1. For Hueske et al. [2012]., we enabled annotation of read- and write-sets, but disabled reordering of DAG-shaped plans.

Optimization time

The time needed to optimize a given data flow with SOFA depends heavily on the number of contained operators. For our evaluation queries, SOFA needed between 0.5 (Q6) and 14 seconds (Q3) to analyze and optimize the respective data flows. During optimization, most time is spent on the Datalog-based reasoning along relationships in Presto. However, time needed for optimization pays off quickly for data analytics tasks at large scale. In all tested data flows, the time spent on optimization amounts to a very small fraction of the time needed for executing the actual flows. For example, the

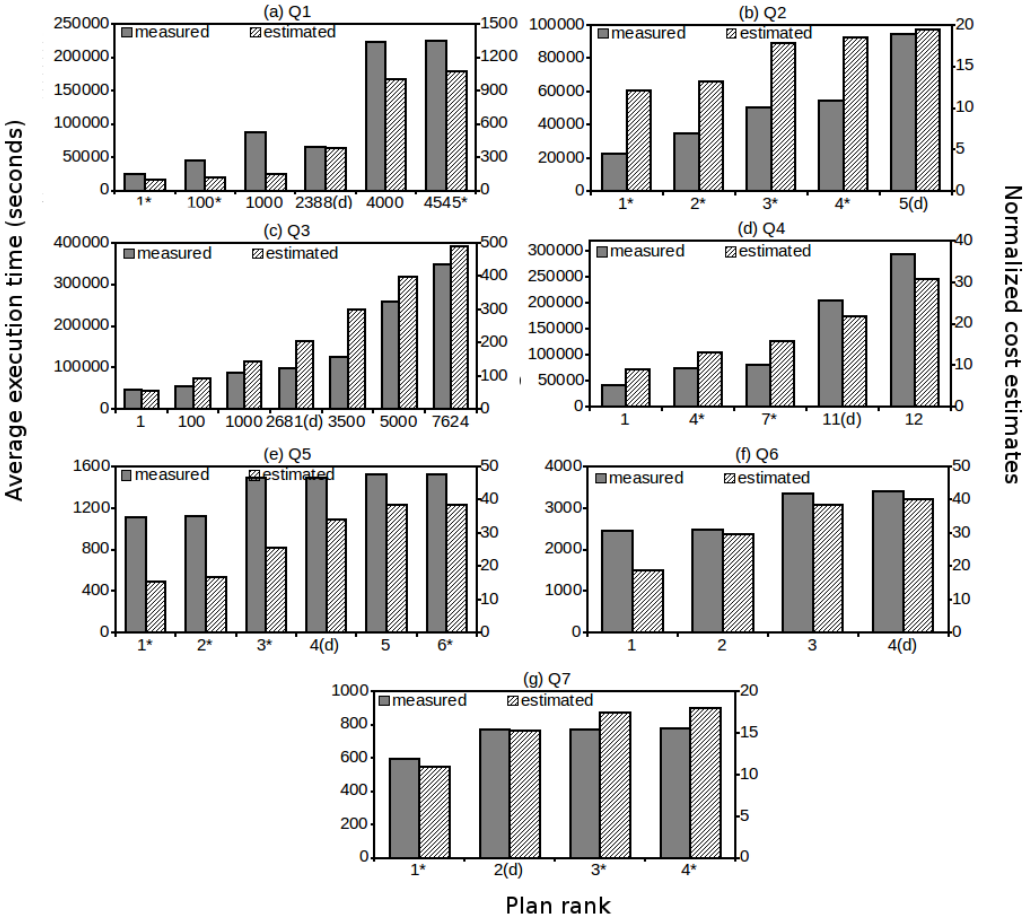


Figure 5.9: Estimated costs (right y axis) and observed execution times (left y axis) of selected plans ranked by cost estimates. Ranks marked with a '*' denote plans found only with SOFA, ranks marked with '(d)' point to the time required by executing the data flows without any optimization.

non-optimized version of Q1 needs more than 18 hours to analyze 100,000 full-text documents, and the optimized version of Q1 analyzes this set of documents in less than 7 hours, whereas SOFA needed roughly 12 seconds to retrieve the best plan.

5.4.1 Finding optimal plans

A large number of semantically equivalent plans for a concrete data flow has the potential to contain the most effective variant. Therefore, we first evaluate SOFA to all three competitors with respect to the number of alternative plans found with each method. We turned search space pruning off and enumerated the complete space of alternative data flows for all queries. In Section 5.2, we explained how complex operators can be resolved into a series of interconnected elementary operators. Q1, Q2, and Q7 contain

complex operators, thus, we enumerated the plan space for these queries both using only elementary operators and using combinations of elementary and complex operators. For the methods presented in [Hueske et al., 2012; Simitsis et al., 2005; Olston et al., 2008], we used complex operators only, as these methods do not provide mechanisms for operator expansion.

As displayed in Table 5.1, SOFA enumerates the largest plan space in all cases. Note that Q1 and Q3 translate to data flows with 10 and 12 operators, respectively, which both contain many degrees of freedom. For example, Q1 and Q3 contain 3 and 5 filter operators (see Appendix D for the concrete scripts). Each filter can be positioned differently in the data flows yielding a high number of alternative plans. The method presented by Hueske et al. is unable to rewrite Q2, Q4, Q5, and Q7, because it is neither capable of rewriting DAG-shaped data flows (Q4, Q5) nor of expanding complex operators (Q2, Q7). The approach of Olston et al. can rewrite only Q3, Q4, and Q6, because these are the only methods that involve filter push-ups. Simitsis et. al find no alternative plans for Q2 and Q7, as in these cases, no adjacent single-input/single-output operators were reorderable. For Q3 and Q6, SOFA and [Hueske et al., 2012] both enumerate the largest plan space, as for both data flows the predominant rewrite options concerned `fltr` operators.

To evaluate the correctness of plan ranking performed by SOFA, we enumerated the complete plan space and ranked the resulting plans ascending by estimated costs for each data flow. We selected and executed differently ranked plans for each data flow and report estimated costs and observed runtimes for these plans.

As shown in Figure 5.9, SOFA ranks the different logical plans correctly, and for Q1, Q2, Q5, and Q7, the best ranked plans were retrieved only with SOFA. We also observed a large optimization potential for most tasks. For example, the best ranked plans for Q1–Q4 outperform the worst ranked plans with factors in the range of 4.2 (Q2) to 9.1 (Q1). For Q5–Q7 we observed differences in execution times of 23% to 28 % between the best and worst plan. Note that these three data flows were the shortest running in our experiments with total runtimes between 10 to 30 minutes, and a significant portion of these runtimes can be attributed to system initialization and communication. Thus, we expect that these data flows benefit much more from optimization on larger data sets. Although we used rather small data sets for evaluating the correctness of the ranking, we see a large impact of choosing a good plan on the overall performance of a data flow. For example, consider Q1 and Q3, where the worst ranked plans were very expensive even for rather small data set of 100,000 full-text articles due to bad placement of expensive operators in the data flows. Specifically, the worst ranked plan for Q1 took more than 2 days to finish and the worst ranked plan for Q3 took more than 4 days to finish, whereas the best ranked plans for these queries were executed in about 6 and 13 hours, respectively.

5.4.2 Pruning

Table 5.1 displays the plan space with search space pruning enabled in brackets. For data flows spanning the largest plan space (Q1 and Q3), pruning helps to significantly reduce the enumerated plan space. For the methods presented in [Olston et al., 2008] and [Simitsis et al., 2005], which both enumerate significantly smaller plan spaces than SOFA, pruning as performed by our enumeration algorithm does not reduce the plan

	SOFA	Hueske et al.	Olston et al.	Simitsis et al.
Q1	**4545 (1032)**	512 (344)	1 (1)	24 (24)
Q2	**5 (5)**	1 (1)	1 (1)	1 (1)
Q3	**7624 (844)**	**7624 (844)**	240 (192)	240 (192)
Q4	**12 (10)**	1 (1)	**6 (6)**	4 (4)
Q5	**6 (4)**	1 (1)	1 (1)	2 (2)
Q6	**4 (4)**	**4 (4)**	2 (2)	2 (2)
Q7	**4 (2)**	1 (1)	1 (1)	1 (1)

Table 5.1: Number of plan alternatives per data flow. Counts in braces denote the number of plans considered with pruning enabled. Bold numbers indicate the plan space containing the fastest plan.

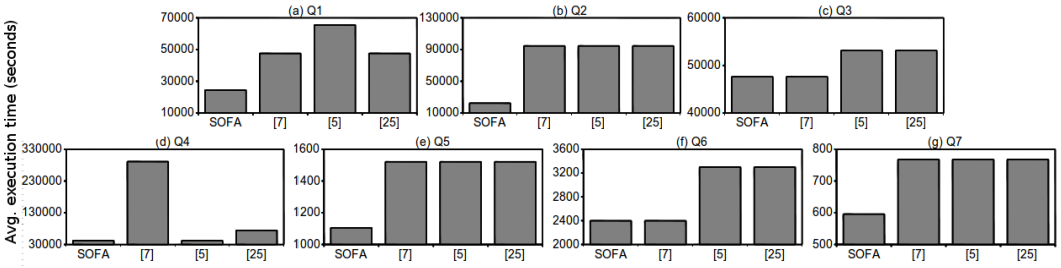


Figure 5.10: Execution times of best plans found with SOFA and best plans found by three competitors.

space in most cases. For each tested data flow, the optimization time with pruning enabled takes not longer than 2.5 seconds with SOFA. Enumerating the complete plan space for each data flow takes at most 10 seconds, which is negligible compared to the execution times of the data flows used for evaluation. Note that the largest part of these optimization times can be attributed to reasoning along Presto relationships, which could be improved using known Datalog optimization techniques [Sagiv, 1987].

5.4.3 Optimization benefits

In our third experiment, we evaluated to which extent data flow optimization benefits from information on operator semantics. Figure 5.10 displays the execution times of the best ranked plan found with SOFA as well as the methods described in [Hueske et al., 2012; Olston et al., 2008; Simitsis et al., 2005]. For each tested data flow, SOFA finds the fastest plan, and for Q1, Q2, Q5, and Q7, SOFA finds significantly faster plans than competitors: the best plan found with SOFA outperforms the best plans found by [Hueske et al., 2012] with factors of up to 6.8 (Q4), and the best plans found by [Olston et al., 2008] and [Simitsis et al., 2005] with factors up to 4.2 (Q2). The method of Hueske et al. performs as well as SOFA for Q3 and Q6, because both methods enumerate the same plan spaces. The rewrite rules of Olston et al. and Simitsis et al. find the same best plan as SOFA for Q4. In these cases, plan optimization involves only reordering filter operators, which is addressed equally well in these methods as in SOFA. Note

	Scale factor	Input size	Optimized (avg. runtime in seconds)	Unoptimized (avg. runtime in seconds)	Gain in %
Q2	1	20 GB	734.44	1,018.26	39
	10	200 GB	5,221.14	6,674.69	28
	50	1 TB	25,057.47	53,934.33	115
	100	2 TB	49,456.58	124,322.50	151
Q6	2	2 GB	175.48	218.60	25
	20	20 GB	225.30	268.16	19
	200	200 GB	674.18	781.97	16
	2,000	2 TB	7,497.36	19,466.59	160
Q7	1	12 GB	237.21	1,113.75	369
	5	60 GB	658.53	4,410.41	570
	10	120 GB	1,190.27	8,679.26	629

Table 5.2: Scalability measurements of optimized and unoptimized plans for selected data flows.

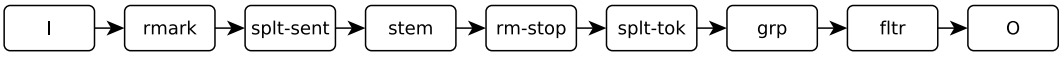
that the method of Hueske et al. cannot rewrite Q4, as this data flow is DAG-shaped. All other data flows involve rewriting general UDFs and expansion of complex operators, and thus, optimization benefits notably from semantic information that is available in SOFA.

5.4.4 Scalability

To evaluate scalability, we executed the unoptimized and optimized data flows for Q2 (Topic detection), Q6 (TPC-H), and Q7 (Entity extraction) on data sets of increasing sizes. Particularly, we manifold the Medline data set we used to evaluate Q2 several times from scale factor 1 (20 GB) to scale factor 100 (2 TB), the TPC-H data set for Q6 from scale factor 2 (2 GB) to scale factor 2000 (2 TB), and the Wikipedia data set from scale factor 1 (12 GB) to scale factor 10 (120 GB). Each data flow was tested on a 12-node cluster with 144 threads and 20GB RAM available on each node. As shown in Table 5.2, data flow optimization as carried out with SOFA is more beneficial the larger data sets grow. Particularly, the optimized plan for Q2 is executed more than twice as fast as the data flow corresponding to the formulated query on 1 TB of input data, whereas on the original data set (20 GB of text data), the optimized plan is 39% faster compared to the unoptimized plan. Similarly, optimizing Q6 achieves a decrease of runtime of 160 percent on 2 TB of input data, compared to 25% of improvement at 2 GB of input data. On Q7, we observe the highest acceleration with factors of between 4.5 on 12 GB and 7.29 on 120 GB of input data, which is due to a possible operator deletion detected by SOFA. The increase of performance gain with larger data sets is due to the vanishing effect of the start-up costs of Stratosphere. These constant costs are responsible for a large fraction of runtimes on smaller data sets, but count less and less the larger the overall runtime of a data flow.

5.4.5 Extensibility

Finally, we concretize the example from Section 5.2 to quantify the effect of pay-as-you-go annotation of operators in SOFA. Recall the novel `rmark` operator, which replaces HTML tags in web pages by a series of '%' of the same length as the removed tags to retain text length and markup position. Imagine a query Q8 that first replaces HTML markup in websites, computes term frequencies from the websites content, and finally filters terms starting with a series of '%'. The corresponding data flow looks as follows:



Initially, `rmark` is annotated only with an *isA*-relationship to the abstract Presto concept *operator*. In this case, SOFA can analyze only read and write access on attributes similar to the method presented in [Hueske et al., 2012], which yields in 10 semantically equivalent plans for Q8. After adding the information that `rmark` is a record-at-a-time operator implemented with a map function, SOFA already finds 18 equivalent logical plans. Finally, when `rmark` is fully specified, including an *isA* relationship to the Base operator `trnsf`, SOFA finds 75 alternative plans.

5.5 User interface

We implemented a web-based interface to enable end-users to pose Meteor queries to the Stratosphere system, which are analyzed and optimized by SOFA and executed by the underlying parallel execution engine. This interface not only highlights SOFA's abilities in optimizing cross-domain data flows with UDFs, but also embraces the entire stack from the Meteor data flow language down to the parallel execution engine Nephele.

Figure 5.11 displays the Meteor user interface. Queries are typed into the text field on the upper left side, which exemplarily displays a query for relationship extraction between persons mentioned in Wikipedia articles. After submitting the query, the translated yet unoptimized Sopremo data flow is displayed in the upper right part of the interface. Boxes depict operators, data sources, and sinks, edges indicate the flow of the data. By clicking on an operator, relevant properties and relationships modelled in Presto can be inspected (cf. Figure 5.12). The bottom of the interface shows a preview of the data to be analyzed (left side) and, after the data flow has been successfully executed, a preview of the result set (right side). During query compilation, users get direct feedback from the system on lexical, syntactical, and semantically soundness of the submitted queries.

Users can experience the entire data flow optimization process step-by-step and visually explore each phase of optimization carried out with SOFA. Figure 5.13 displays the SOFA optimizer interface. The upper part in this window displays the precedence graph determined by SOFA. Plan alternatives are visualized in the bottom part of the interface ranked by estimated costs. A plot in the middle of the interface summarizes estimated plan costs and indicates individual operator costs through a color scheme. Users can inspect any plan alternative together with its estimated costs and select it for execution.

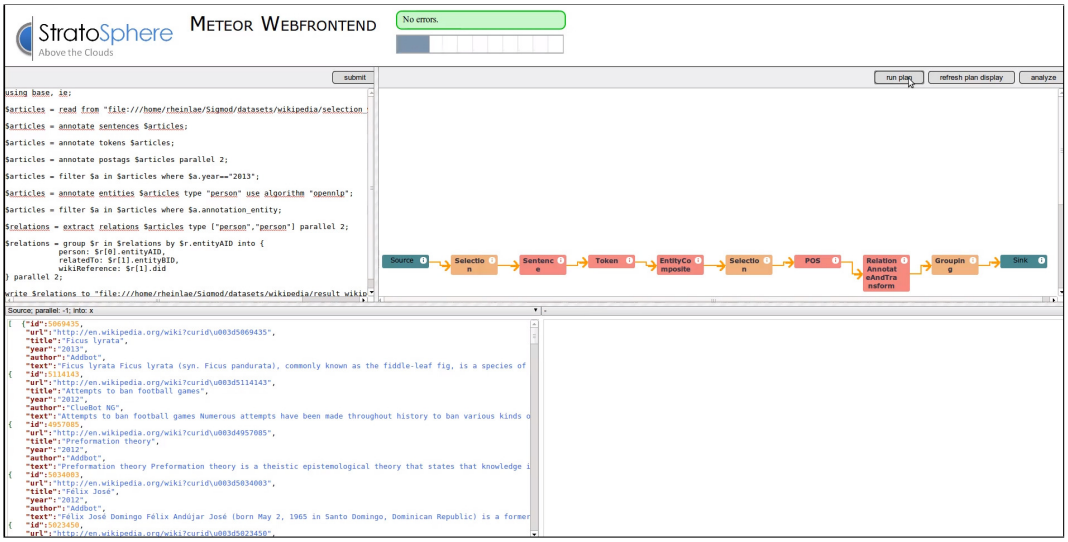


Figure 5.11: Meteor query interface (top left) and Sopremo data flow compiler (top right). The bottom of the figure displays excerpts of the input data to be analyzed (left) and the result set (right).

Once the user selects a logical data flow for execution, it is translated into a parallel PACT program, physically optimized, and submitted to Nephele for parallel execution. A separate interface visualizes the execution of the data flow program, featuring the parallel execution graph together with the color-indicated status of tasks (waiting, running, finished, failed), and information on resource consumption. After the data flow has successfully been executed, a preview of the result set is available in the bottom right part of the interface shown in Figure 5.11.

5.6 Summary

In this chapter, we addressed the problem of semantics-aware optimization of data flows with UDFs and presented SOFA, a novel, extensible, and comprehensive optimizer. SOFA builds on a concise set of properties describing the semantics of Map/Reduce-style UDFs and a small set of rewrite templates to derive equivalent plans. SOFA optimizes logical data flows, which can be compiled into physical data flows consisting of parallelization functions.

A unique characteristic of our approach is extensibility: we arrange operators and their properties into taxonomies, which eases integration and optimization of new operators. Our experiments reveal that SOFA is able to reorder acyclic data flows of arbitrary shape (pipeline, tree, DAG) from different application domains, leading to considerable runtime improvements. We also show that SOFA finds plans that outperform found by other techniques. SOFA was implemented on top of the Stratosphere system, however, our approach is equally applicable to other parallel data analytics systems that build upon such data flows, in particular those using the Map/Reduce paradigm as implemented in Hadoop. For example, Pig [Olston et al., 2008] compiles a query into a

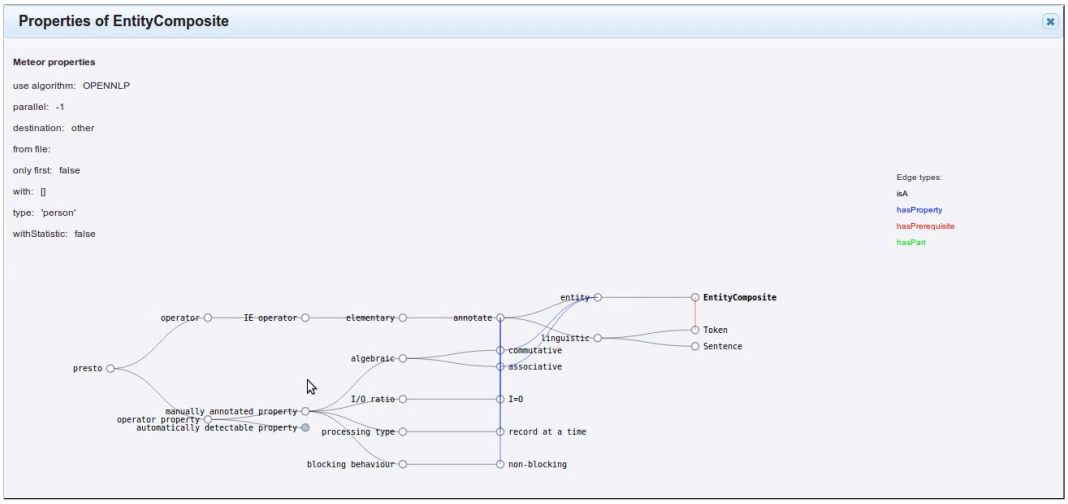


Figure 5.12: Presto graph explorer showing an exemplary subgraph for a complex entity extraction operator.

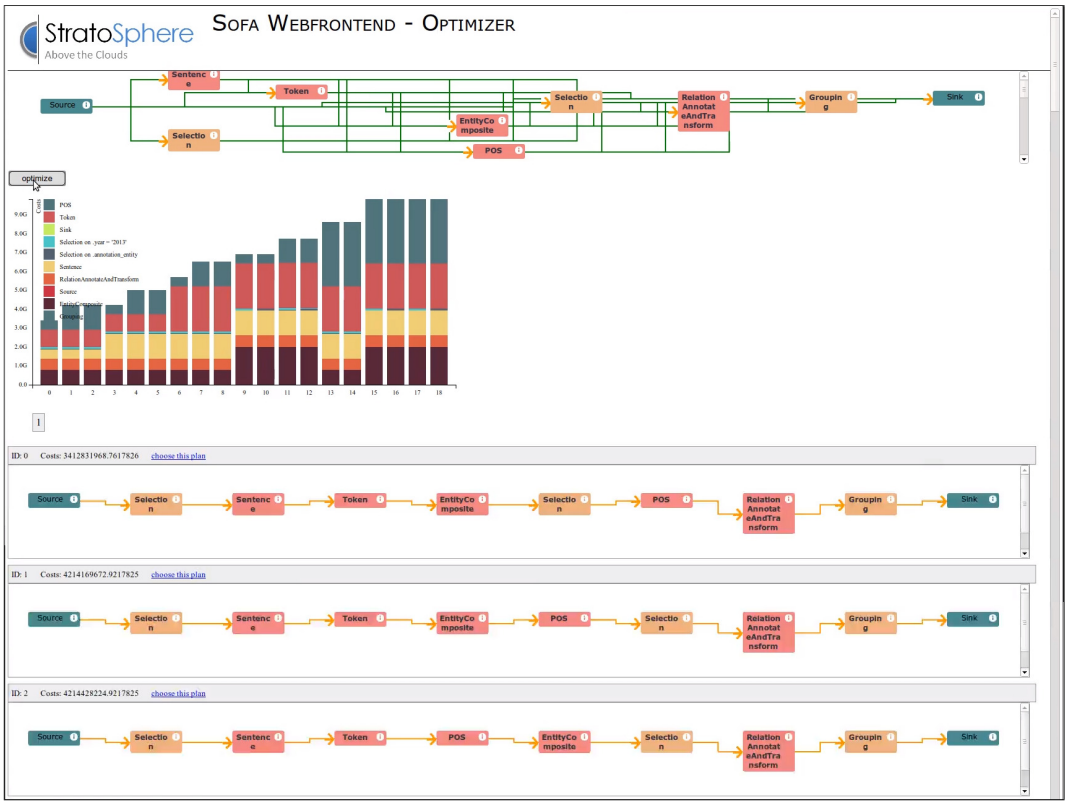


Figure 5.13: SOFA data flow analysis interface showing the precedence graph (top), cost estimates (middle), and plan alternatives for the data flow from Figure 5.11.

logical operator plan, which is translated into a physical data flow consisting of map and reduce parallelization functions. Similar to Stratosphere, SOFA could optimize such a data flow based on inferred or annotated properties. The parallelization functions used in PIG/Hadoop are a subset of the functions supported by Stratosphere and therefore already accounted for during optimization. Thus we believe that our contributions here can also be fruitful for other parallel data analytics systems.

6 Domain-specific information extraction at web scale

In this chapter, which was published in [Rheinländer et al., 2016], we report our experiences from building a system for domain-specific text analytics on the open web using the techniques presented in the previous chapters 2–5. The Stratosphere system with its operators for scalable and declarative IE and WA is now used to analyze a very large, real world data set of 1 TB of unstructured documents crawled from the open web.

The analysis of information published on the web has been shown to be valuable for a plethora of applications, for example, to analyze customer product reviews [Pang and Lee, 2008], to investigate relationships between politicians and their sponsors [Heise and Naumann, 2012], or to predict flu waves and assess their treatments [Covolo et al., 2013], to name just a few. In this chapter, we intend to study another open research question, namely, whether valuable biomedical information, which may augment or dissent facts published in scientific publications, is available on the open web and how such information is best extracted and distilled for in-depth analysis by domain experts using a system for large-scale IE inside a parallel data analytics system.

Analyzing web data is not trivial due to its scale, distribution, heterogeneity, redundancy, and a questionable quality of the information posted online. Compared to traditional text analytics, already obtaining the data to be analyzed is difficult, requiring either access to an existing large web crawl or the setup and running of a proper crawler. For applications requiring domain-specific texts, like the one we focus on here, special care must be taken to restrict the crawl to this domain, typically by applying text classification on the crawl or during the crawling [Chakrabarti et al., 1999; Davison, 2000].

Another severe issue arises from the extreme heterogeneity of web documents and their cluttering with noise and errors, for example, navigational elements, advertisements, meta data, script code, formatting instructions etc. [Yi et al., 2003]. In fact, such elements constitute the largest parts of HTML documents. For instance, only a small fraction of the tables contained in HTML documents contains meaningful relational information [Cafarella et al., 2008]. Many projects circumvent this problem by focusing on a single or a few well-known and well-structured data sources, typically the big social media platforms such as Twitter, Facebook, or FlickR; however, this excludes literally billions of additional knowledge sources. Furthermore, the filtered and cleansed web texts must be analyzed by IE algorithms to obtain the desired facts, which in itself is a challenging task when the text collection is large and the requirements regarding data quality are high.

Building comprehensive systems for domain-specific text analytics on the open web for long was only possible for large web companies; however, advances in cloud computing, information extraction, and crawler techniques together with falling prices for storage, computing power, and network bandwidth put such systems – in principle –

also into the realm of mid-size organizations. But putting this theoretical possibility into practise still is a highly challenging task. Therefore, the goal of this chapter is not only to describe such a system, with a focus on design issues regarding robustness, data quality, and scalability, but also to pinpoint the most critical issues that need to be solved when building such systems with the ultimate intention to foster more research into this important and challenging area.

We present a case study on extracting biomedical information from the web by means of the parallel data analytics system Stratosphere and the state-of-the-art IE operators we developed for this system (cf. Chapter 3). For data collection, we first customized and applied Apache Nutch[22] to crawl a 1 TB collection of web text from the biomedical domain with the goal to retrieve a high quality corpus in terms of precision with respect to our target domain. This corpus was cleansed and filtered by specific Stratosphere modules for web texts, linguistically preprocessed using methods from statistical natural language processing (NLP), and eventually analyzed by a series of domain-specific IE programs to find mentions of important biomedical entities, such as genes, drugs, or diseases. We then ran the same pipeline on two much more controlled sets, i.e., all abstracts in the Medline collection of scientific articles, and a set of approximately 250.000 biomedical full texts. A fourth corpus was built from all web pages deemed out-of-domain by the focused crawler. Next, we compared results from a linguistic analysis and from the domain-specific IE on the four corpora to each other, finding notable differences in many aspects, including simple metrics such as average sentence and document length, more linguistically motivated properties such as the use of negation or abbreviations, and, eventually, the sets and frequencies of occurring domain-specific entities. The system applies advanced machine learning in every phase of its collection and analysis pipeline, i.e., text classification during focused crawling, snipped classification for the extraction of net text from HTML pages, sequential classification with Hidden Markov Models for NLP, and classification, pattern matching and Conditional Random Fields for IE tasks.

The entire process for web text analysis (excluding crawling) was specified, optimized, and executed using a small set of data flows in Stratosphere [Alexandrov et al., 2014], which allowed us to evaluate the entire extraction process with respect to scalability, efficiency, and quality of the involved tools. We believe our approach implements a notable advancement compared to the current state-of-the-art for building such systems, which boils down to manually created scripts implementing an ad hoc assembly of existing tools. This practice clearly interferes with today's needs in Big Data analytics; instead, we envision complex information acquisition and extraction from the web as an almost effortless end-to-end task.

The remainder of this chapter is structured as follows: We first describe the acquisition of biomedical web documents by means of parallel focused crawling and we present details on proper seed generation and document classification to obtain a large data set of reasonable quality. Second, we discuss the analytical data flows we used for analyzing the crawl regarding to graph structure, language structure, and biomedical entity extraction. We evaluate these data flows with respect to scalability and efficiency of the involved algorithms. We deeply analyze in total 1 TB of crawled documents in terms language structure and biomedical contents and present the first comprehensive char-

[22] `http://nutch.apache.org` (last accessed: 2016-10-05)

acterization in which way biomedical web documents differ from biomedical articles and abstracts published in scientific journals. Our analysis suggests that extracted information from biomedical web texts has valuable potential to augment knowledge contained in biomedical databases. Finally, we summarize lessons learned in this study and highlight open engineering and research challenges for efficient text analytics at large scale.

6.1 Corpus generation by means of focused crawling

The goal of our research is to perform advanced IE on domain-specific collections of web documents. A proper way to obtain such a collection is to perform a focused web crawl, where crawler automatically traverses parts of the web to find documents relevant for a certain topic [Davison, 2000]. To speed-up the crawling and to obtain good harvest rates (i.e., a large density of relevant pages among all crawled pages), a major objective during focused crawling is to visit only those outgoing links of a website that appear to be particularly relevant for a given topic. To decide whether a link is relevant or not, it is commonly assumed that relevant pages are most likely linked to other relevant pages whereas irrelevant pages point more often to other irrelevant pages and thus constitute an endpoint during the crawl [Olston and Najork, 2010]. This assumption is exploited during focused crawling such that only those websites are visited, which are linked to a relevant node. To assess the relevance of page, a focused crawler is equipped with a text classifier trained on a set of pre-classified documents. We built a focused crawler which pursues the following approach: It downloads web pages, classifies them as relevant or not, and only considers links outgoing from relevant pages further. We did not follow the alternative approach of classifying links based on its surroundings because this would require the laborious creation of a training corpus of links; in contrast, obtaining a training corpus of relevant documents is comparably simple. For our study, we trained on a set of randomly selected abstracts from Medline[23], considered as relevant, and an equal-sized set of randomly selected English documents taken from the common crawl corpus[24], considered as irrelevant. This approach is cheap and simple; note, however, that it introduces some bias as a typical Medline abstract is quite different from a typical web page (see Section 6.3).

6.1.1 Crawler architecture

To obtain a large corpus of reasonable quality, the setup of the focused crawler is crucial. Web crawlers should be insusceptible to so-called spider traps, i.e., websites containing programming errors or dynamically generated links that cause the crawler to be trapped in an infinite loop. A crawler also needs to respect the implicit and explicit rules of a domain (e.g., maximum number of simultaneous requests, rules contained in the "robots.txt" file). Finally, it must be implemented in a distributed manner to allow for using multiple machines in parallel. There exists a number of frameworks which implement such functionality; we built our system on top of the open-source framework

[23] `http://www.ncbi.nlm.nih.gov/pubmed` (last accessed: 2016-10-05)
[24] `http://www.commoncrawl.org` (last accessed: 2016-10-05)

Apache Nutch, which is based on Apache Hadoop[25] to enable scalable and distributed crawling. It lacks a component for focusing a crawl, but has a clean extension interface which we used to plug-in a classifier and the necessary logic.

Figure 6.1 shows the architecture of Nutch together with custom extensions we integrated to enable a topical crawl focusing on the biomedical domain. The part implemented in Nutch (cf. upper part of Figure 6.1) is fairly conventional; an injector reads seed URLs from a text file and adds these to the crawl database (CrawlDB). CrawlDB acts as a frontier and stores all information necessary for a certain URL (e.g., fetch status, meta data). The generator component creates lists of yet unvisited URLs ("fetch lists") that are processed by multiple fetcher threads in parallel. A set of fetcher threads reads lists of yet unvisited URLs from CrawlDB, connects to the respective servers, downloads the web pages represented by the URLs, and stores them as segments ("data shards"). Each downloaded page is forwarded to the parser component, which extracts outlinks, meta data, and the main textual content of the page. Unseen links are added with the status "unfetched" to CrawlDB and visited URLs get the status "fetched". Besides, a link database (LinkDB) stores all incoming links for the given URLs and thus represents the web graph traversed during crawling. LinkDB is updated with newly extracted outgoing links and finally, newly created segments are indexed.

To add focus to the crawling process, we extended Nutch with the following components (cf. lower part of Figure 6.1): After parsing a web page, we first check whether it is of textual content using a MIME type filter. If a page passes this filter, a pre-selection component checks if a page is suitable for further investigation. This phase involves a length filter that rejects pages shorter or longer than a certain threshold and an n-gram based language filter for filtering out all non-English language texts, because subsequent IE tools are sensitive to language (cf. Chapter 2). Afterwards, the main text of a page is extracted using the tool Boilerpipe [Kohlschütter et al., 2010]. To classify a document, its extracted net text is segmented into tokens, stop words are removed, and all remaining tokens are converted into lower-case. Finally, we create a bag-of-words model from all remaining tokens of a document to enable classifiaction with respect to biomedical relevance.

We use a Naïve Bayes algorithm due to its robustness with respect to class imbalance (we have no rational guess on the expected percentage of biomedical pages during a focused crawl) and its ability to update its model incrementally, although we currently don’t use this feature. If a page is classified as relevant, it is added to the corpus and all its outlinks are added to CrawlDB. Otherwise, if a document is identified as irrelevant either during pre-selection or classification, an update process deletes all outgoing links from this page from CrawlDB. The crawling and classification process is repeated iteratively until either CrawlDB is empty, the desired corpus size is reached, or it is stopped manually by the user.

6.1.2 Seed generation

A very important issue in crawling, and especially in focused crawling, is to determine the set of seed URLs used to initiate the crawl. The typical way of obtaining a large set of seeds is to issue keyword queries to one or more search engines. For focused

[25] `http://hadoop.apache.org` (last accessed: 2016-10-05)

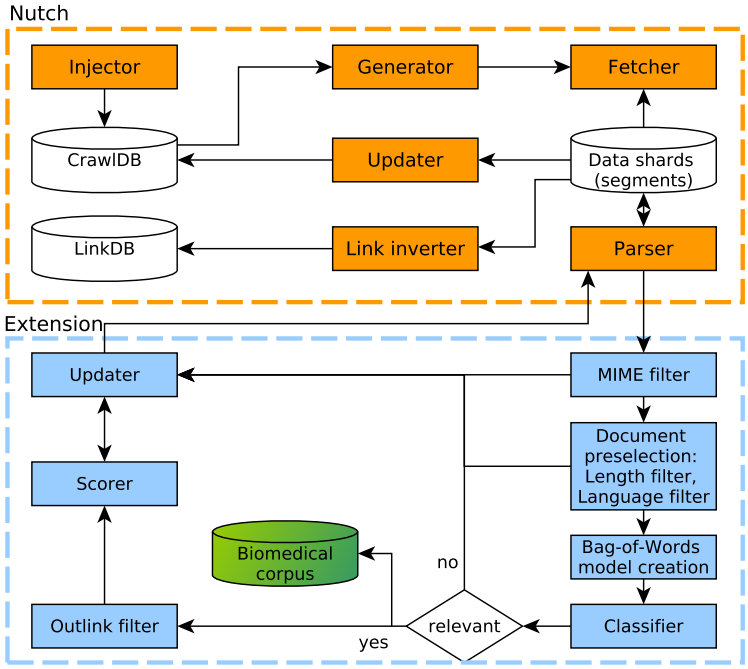


Figure 6.1: Architecture of a topical crawler based on Apache Nutch.

crawling, keywords are chosen such that they retrieve domain-specific seeds with high probability. Since all search engine APIs restrict the number of allowed queries and limit the number of returned results, one often uses (a) multiple search engines in parallel and (b) large sets of queries - which creates the necessity to generate thousands of high quality queries. For our case study, we utilized five different search engines, namely Bing[26], Google[27], Arxiv[28], Nature[29], and Nature blogs[30].

For each search engine, we generated queries with (a) general biomedical terms, obtained from National Cancer Institute[31] and the Genetic Alliance glossary[32] and (b) highly specific molecular terms extracted from the Gene Ontology[33], Drugbank[34], and the UMLS/MeSH sub-tree for diseases [35]. Exemplary keywords are shown in Table 6.1 together with the total number of search terms for each category. Clearly, the chosen queries give the resulting corpus a certain direction; in our case, we intended to focus on genetic facts about diseases and possible treatments.

In a first experiment, we used only a subset of keywords from our data sources (see Table 1, numbers in bracket). All search results from the different search engines ob-

[26] `http://www.bing.com` (last accessed: 2016-10-05)
[27] `http://www.google.com` (last accessed: 2016-10-05)
[28] `http://www.arxiv.org/find` (last accessed: 2016-10-05)
[29] `http://www.nature.com/search` (last accessed: 2016-10-05)
[30] `http://www.blogs.nature.com/` (last accessed: 2016-10-05)
[31] `http://www.cancer.gov` (last accessed: 2016-10-05)
[32] `http://www.geneticalliance.org.za/resources/glossary.htm` (last accessed: 2016-10-05)
[33] `http://geneontology.org` (last accessed: 2016-10-05)
[34] `http://www.drugbank.ca` (last accessed: 2016-10-05)
[35] `http://www.nlm.nih.gov/mesh/` (last accessed: 2016-10-05)

Category	No. of terms	Example search terms
general terms	500 (166)	cancer, chronic pain, gene expression, symptoms
disease-specific	5000 (468)	acne, cough, diarrhea, nausea, thymoma
drug-specific	4000 (325)	aspirin, claforan, estraldine, GAD-67, prednisone
gene-specific	6500 (246)	ATXN, BRCA, cactin, interleukin, protocadherin

Table 6.1: Total number of search terms and example terms by category used for seed URL retrieval. Numbers in brackets denote the number of search terms for the first crawl (see text).

tained with these keywords were merged to a single list of 45,227 seed URLs. However, the resulting crawl terminated quickly due to an emptied CrawlDB, i.e., all pages in the frontier of pages reachable from these seeds were classified as irrelevant. Debugging this crawl, we found several reasons for this situation. First, the search terms chosen were too general. For these, the search engines return rather general pages, which they considered as authorative for the respective topic, such as front pages of portals. These pages were very often immediately classified as irrelevant, i.e., this branch of the crawl stopped after just one step. Second, we found that biomedical sites generally are only weakly linked; most often, all outgoing links from a page were navigational leading to pages on the same host (see Section 6.3.1 for details). Note that not stopping the crawl of such irrelevant pages immediately but after n steps (e.g., $n = 2$, $n = 3$) is a viable alternative to increase the size of the crawl, since many front pages of portals deemed irrelevant link to relevant content for our domain. Yet, crawling time will significantly increase since many irrelevant pages will be explored as well. Since the entire crawling process already stretched to more than two months while stopping immediately when visiting an irrelevant page (see Section 6.3.1 for details), we decided to increase the set of seed URLs to obtain a larger crawl instead. Consequently, we performed a second seed generation run, this time using 15,000 queries resulting in a total number of 485,462 seed URLs, which were used for crawling.

6.2 Data flows for web-scale IE

The ultimate goal of our study is to compare the domain-specific information content of biomedical web pages with that of a corpus of scientific publications, either as full texts or as abstracts. Furthermore, we want to compute general linguistic characteristics of the different corpora, such as average sentence length or frequency of negated sentences. The latter has two purposes: First, we find it of general interest to differentiate properties of different types of texts in a given domain. Second, these properties have consequences on the tools which are used to analyze the texts. Eventually, we are also interested in the link topology of the biomedical pages to test some common assumptions on these structures.

Processing a large set of documents to compute a number of measures and to extract a variety of different information requires a complex set of tasks. Specifically, our scenario encompasses three different sub-problems, namely document preprocessing and

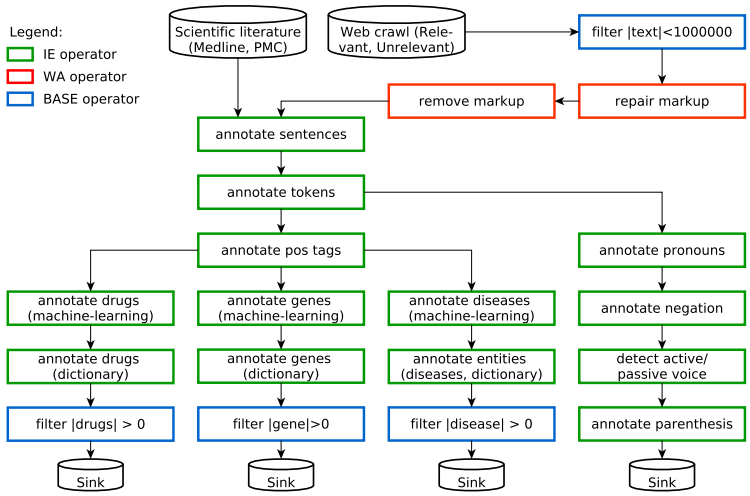


Figure 6.2: Consolidated high-level data flow for analyzing biomedical documents from scientific publications and the internet. Isochromatic frames indicate Sopremo operator package.

HTML cleansing, linguistic document analysis, and biomedical entity extraction. Each of these subproblems requires the use of a variety of different tools; for instance, high-quality entity extraction is only possible using separate and highly optimized tools for each entity class. Furthermore, the different processing steps have various dependencies. Ideally, all tasks and their dependencies are expressed using a single data flow program to enable a holistic optimization and seamless analysis. Stratosphere allows such a concise specification as detailed in Chapters 2, 3, and 5. Figure 6.2 displays the high-level data flow we developed for this case study, which contains in total 19 complex operators that compile down to 38 elementary operators.

Web pages are first filtered to exclude extremely long documents from the analysis. Next, HTML markup is detected, errors are repaired, and all markup is removed from each page. Links are extracted to enable an analysis of the web graph of the crawl. All documents are annotated with sentence and token boundaries. For linguistic analysis, each sentence is analyzed for occurrences of pronouns, negation, passive voice, and parenthesis using different sets of regular expressions, and each found mention of any of these categories is added to the result set together with information on document ID, sentence ID, and start/end positions. For the biomedical content analysis, we then annotate three types of biomedical entities, namely genes, drugs, and diseases.

Note that biomedical IE is considered particularly challenging because of ambiguous naming conventions, multi-word names, acronyms, etc., and specialized methods, dictionaries, and models are needed to achieve satisfactory results with respect to extraction accuracy [Leser and Hakenberg, 2005]. The accuracy of all tools applied to web documents is difficult to predict, since all current tools were trained and evaluated only on scientific articles, and mostly only on abstracts. Therefore, we chose to apply two different extraction methods for each entity type on the entire data set: A classical fuzzy dictionary-matching tool, and ML-based entity taggers using Conditional Random Fields (CRF). In this field, dictionary-based entity extraction typically achieves good

precision yet low recall because dictionaries are necessarily incomplete in a field developing as fast as biomedical research. On the other hand, ML-based extraction methods often show much improved recall and they also show superior precision. Besides, they usually exhibit a significantly slower processing speed.

Note that none of the tools we applied was developed for this study; instead, they were chosen from available open source software following a best-of-breed strategy (cf. Chapter 3). For dictionary matching, we adapted an automaton-based matching algorithm originally developed for species recognition that quickly retrieves mentions of entities even for large dictionaries [Gerner et al., 2010]. To account for some variations, we transformed each dictionary term into a regular expression. The largest dictionary employed here, that for gene names, contains more than 700,000 entries; dictionaries for disease and drug names were significantly smaller with 61,438 and 51,188 entries, respectively. As ML-based tool for drug names we used ChemSpot [Rocktäschel et al., 2012], for gene names we applied BANNER [Leaman and Gonzalez, 2008], and for diseases we integrated a previously developed tool in our group directly building on the CRF library Mallet[36]. Note that also ChemSpot and BANNER use Mallet.

Once entities and linguistic structures are extracted from web documents, a main interest in this study is to compare these results to the scientific literature to identify commonalities and differences between biomedical texts from the web and from peer reviewed journals. Thus, we also analyzed abstracts and full-texts from Medline[37] and Pubmed Central (PMC)[38], the largest collections of freely available biomedical citations (Medline) and full-texts (PMC), with the same data flow. Since these documents are not in HTML format but given in plain text, markup repair and removal is not performed.

6.3 Evaluation

In this section, we evaluate different facets of our case study in domain-specific web-scale information extraction. Our analysis encompasses three different dimensions: First, we first give a technical evaluation of the focused crawl itself, which implies assessing the harvest rate of the crawler, the quality of the text classifier used to keep focus, the quality of the boilerplate detection algorithm we applied, and a structural analysis of the crawled data to gain insights into the interconnectedness of biomedical web pages and to identify possible authoritative and leading domains. Second, we focus on the performance and scalability of the information extraction pipeline, which includes identifying the most time-consuming steps, contrasting dictionary-based entity recognition with ML-based approaches in terms of recognition speed, and experiments for showing the scale-out our pipeline achieves based on the underlying data analytics system Stratosphere. Third, we provide a content analysis of our biomedical web corpus in contrast to three other corpora, i.e., the set of pages classified as irrelevant during the crawl, the complete set of approximately 21 million abstracts from Medline (until year 2013), and a set of approximately 250,000 full texts from the PLoS open data mining collection (PMC). This content evaluation encompasses a linguistic analysis concerning article lengths, sentence structure and usage of grammatical structures to

[36] `http://mallet.cs.umass.edu` (last accessed: 2016-10-05)
[37] `http://www.ncbi.nlm.nih.gov/pubmed` (last accessed: 2016-10-05)
[38] `http://www.ncbi.nlm.nih.gov/pmc` (last accessed: 2016-10-05)

learn whether the language used in the biomedical web is structurally different from the language used in peer-reviewed biomedical publications, and we analyze mentions of biomedical named entities and compare these to what is known from the literature to assess both the quality of our crawl and the overlap with the biomedical literature.

6.3.1 Quality of the focused crawler

The premise of our study is to enable seamless information extraction on large sets of crawled, domain-specific documents on a mid-size cluster. We run our crawler on a cluster of 5 servers, each equipped with at least 32 CPU cores and connected with a 1 GB line to a 10 GB switch connected to the Internet backbone. Politeness rules of web servers were respected and the sizes of host-specific fetch lists was limited to 500 to prevent threads from blocking each other. Each downloaded page passing the initial filtering was subjected to boilerplate removal and to text classification. With this setup, our crawler achieved a download rate of 3-4 documents per second, which is notably slower compared to other systems (e.g., Olston and Najork [2010] consider download rates between 10 and 100 pages per second as representative) due to the complex filtering and classification steps employed in our setup. This sums up to more than 80 days of pure crawling and classification for downloading and analyzing approximately 21 million web pages. The crawl yielded 373 GB presumably relevant and 607 GB presumably irrelevant pages, which corresponds to a harvest rate of 38%. This seems to be a typical value for such systems (e.g., Chakrabarti et al. [1999] and Pant and Srinivasan [2005] report harvest rates between 25% and 45%). Document pre-selection was very effective: MIME-type filtering decreased the number of documents to be analyzed by 9.5%, language filtering by 14%, and document length filtering by 17%.

Evaluating a focused crawler is notoriously difficult for multiple reasons. First, experiments cannot be repeated due to the highly dynamic nature of the web. Starting a crawl with exactly the same set of seeds will result in a largely different result even if the repetition is performed shortly after the first run, as many pages will have changed leading to different link chains. Due to this fact, one cannot easily compare the performance of, for instance, two crawls using different classifiers, different relevance thresholds, or just different prioritization rules for the fetch queue. Second, the recall of a crawler cannot be determined; even estimating it is impossible as this would require a certain set of pages one expects to be found; but whether or not a crawler finds them largely depends on the seeds which cannot be set in an "unbiased" fashion. Third, yield and harvest rate depend largely on the seed lists, which usually are not published.

Classifier and boilerplate detection

The quality and size of the crawled corpus for our purposes, i.e., its specificity for the biomedical domain, depends on mostly two factors: The quality of the classifier, the size of the seed list, and the quality of the boilerplate detection. We assessed the quality of both components on a gold standard data set during development and on a small, randomly drawn sample of the crawl.

Our classifier achieved a precision of 98% at a recall of 83% in 10-fold cross validation on its training corpus. We then manually checked a randomly drawn set of 100 pages from the relevant corpus and 100 pages from the irrelevant corpus. On these 200 pages,

about.com	arxiv.org	bettermedicine.com
biomedcentral.com	blogger.com	blogs.nature.com
cancer.net	cancer.org	cdc.gov
definition-of.com	disqus.com	farlex.com
healthline.com	hhs.gov	lexiophiles.com
mpg.org	mypacs.net	g2conline.org
omniture.com	ourhealth.com	reuters.com
rightdiagnosis.com	sideeffects.embl.de	slideshare.net
statcounter.com	thefreedictionary.com	nih.gov
wikimedia.org	wikipedia.org	wordpress.org

Table 6.2: Domains of 30 top-ranked sites according to page rank.

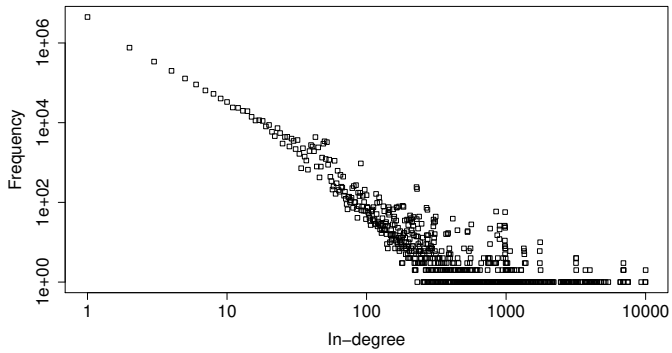
precision was estimated at 94% at a recall of 90%, which roughly confirms the results on the training data (note that these are quality measures of the classifier, not of the the entire crawler; see discussion above). Differences are notable, but in expectable ranges gives the different characteristics of the texts and the small sample size. An analysis of the false positives showed that these are often web pages at the fringe of what we consider biomedical; for instance, pages describing chemical support for body builders or technical devices used for medical purposes such as wheel chairs. Note that the classifier model we used is geared towards high precision as classifier recall plays a minor role in focused crawling; assuming that the web is essentially infinite, one can simply let the crawler run for longer to obtain more relevant documents.

In an initial evaluation on a gold standard data set, the boilerplate detection tool we used achieved a precision of 90% at a recall of 82% on average, evaluated on a set of 1906 web pages. These quality measures are computed based on the amount of net text being correctly identified by the algorithm. We assessed the quality of the method on the same 200 web pages used for judging the text classifier. Results indicate a precision of 98% at a recall of 72%. Manual inspection revealed that tables and lists, which often contain valuable facts, are not recognized properly in many cases.

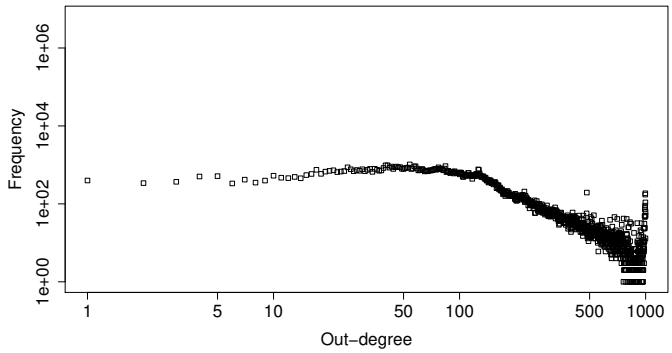
Table 6.2 lists the top 30 ranked domains according to page rank [Page et al., 1999]. Manual inspection revealed that many of them clearly relate to biomedical content, which suggests that the crawling process points to our target domain. Sub-classes of seemingly irrelevant sites, such as slideshare.net or blogger.com often also contain some biomedical material (e.g., blogs, personal journals, reports). It is also not surprising that domains such as arxiv.org and nature.com are ranked within the top 30, because seeds were generated by the search APIs of these domains, which return results only for content hosted there.

Graph structure of the crawl

To analyze structural aspects of our crawl, we first determined the occurrences of `<a href>` tags on the relevant crawled pages and computed distributions of in- and out-degrees in this data set. It is known from the literature that distributions of in- and out-degrees in web graphs tend to follow a power-law distribution, i.e., many nodes have small in-degrees whereas only a few nodes have a large in-degree [Broder et al., 2000]. Figure 6.3 shows the in-degree and out-degree distributions of relevant web



(a) In-degree distribution (log-log-scale)



(b) Out-degree distribution (log-log-scale)

Figure 6.3: Distribution of in- and out-degrees of relevant crawled data.

pages. Clearly, the in-degree distribution of our data also follows a power-law, which confirms previous results and indicates that there are rather few authoritative websites within the biomedical domain [Kleinberg, 1999]. On the other hand, the out-degree distribution does not follow a power-law, as many pages have a rather high out-degree. We believe that this is a property of the biomedical domain for the following reasons: First, many relevant pages follow the template of a scientific article, with ample reference to other articles or web sites. Second, there exist hundreds of public databases of biomedical entities to which many web pages refer to when discussing these entities. Moreover, a large proportion of the links on a web page can also be accounted for by internal references on the same page or navigational elements.

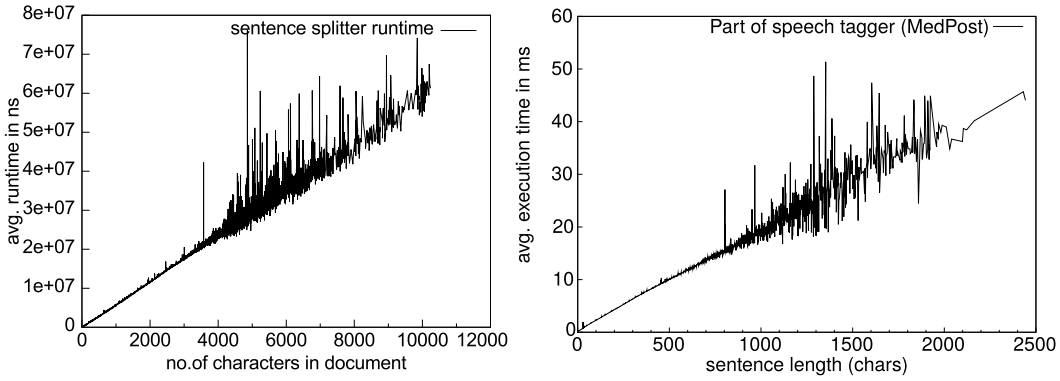
6.3.2 Scalability of IE

We evaluated the performance of the extraction and analysis data flow in terms of scalability and performance of the individual IE components. These experiments were carried out on a 28 node cluster, where each node was equipped with 24 GB RAM, 1 TB HDD, and a Intel Xeon E5-2620 CPU with 6 cores. Accordingly, the maximum degree of parallelism (DoP) was 168. In the following, we always report as runtimes the average of 3 runs of the analysis flows on each corpus. Input and output of all tasks was stored in HDFS with one data node per compute node and a data replication factor of 3.

Runtime characteristics of the different IE tools

All NLP and IE tools available for Stratosphere were originally designed and implemented by third parties. Many of them are complex applications encompassing several thousand lines of code with multiple dependencies to external libraries. This implies that we usually have no influence on the speed or memory consumption of these tools; there are very rare cases where command-line parameters can be used that impact these properties. Of course, we do heavily influence the speed of each tool on the entire data set by parallelizing its execution on different partitions of the data set.

Prior to analyzing the entire data set of crawled documents, we first evaluated the individual runtimes of each involved component using a random sample of 10,000 documents, which were analyzed using a single thread on a single server. The two dominant steps with respect to runtime are entity extraction, consuming 70% of the total execution time, and part-of-speech tagging, requiring 12% of the runtime. The distribution of the runtimes of sentence splitting(a) and part-of-speech tagging (b) are shown in Figure 6.4 and the runtimes of dictionary and ML-based entity annotation are shown in Figure 6.5. Two observations are particularly interesting. In principle, the tagger's and the sentence splitter's runtime is linear in the length of the text being analyzed. Particularly, our part-of-speech tagger, MedPost, uses a Hidden Markov Model of order three. There are, however, large runtime fluctuations in practice (see Figure 6.4(b)) and even occasional crashes, especially when the splitter and the tagger are applied to very long sentences. Clearly, it is highly questionable whether the very long sentences we observe in our data (with more than 2000 characters) are really reasonable sentences or just errors of the sentence detection method; however, such errors are inevitable in a web environment, considering that the input to the splitter are parts, possibly wrongly extracted by the boilerplate detection, of arbitrary web pages possi-



(a) Sentence splitting

(b) Part-of-speech tagging

Figure 6.4: Runtimes of linguistic analysis tools with respect to the length of the input texts.

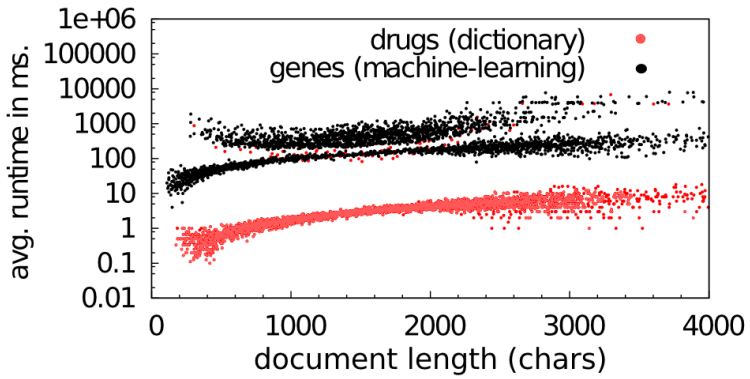


Figure 6.5: Runtimes of NER tools with respect to the length of the input texts. Black: ML, red: dictionary.

bly without any sentence structures (see also Section 6.5). One work-around would be to introduce an upper limit on sentence length, but finding a good threshold, trading runtime robustness for information yield, is non-trivial. Second, Figure 6.5 shows that the execution time needed for annotating entities varies greatly between annotation methods. Dictionary- and ML-based methods differ in runtime by up to three orders of magnitude. This is a consequence of the differing computational complexity of the underlying algorithms; essentially linear for dictionary matches (the regular expression transformations almost only affect very short word suffixes), yet quadratic for the Conditional Random Fields underlying our ML-based tools [Viterbi, 2006].

Scalability

We tested the scalability or our IE data flow using a random sample of 20 GB from our crawl. Experiments were carried out separately for the linguistic analysis and the

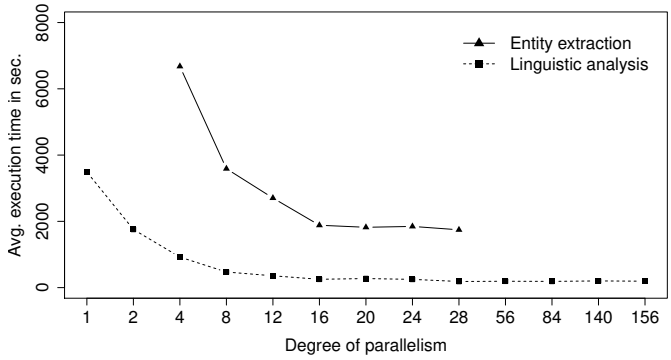


Figure 6.6: Scale out of linguistic and entity extraction data flows.

biomedical entity annotation to gain insights into their specific behavior. To this end, we created two separate data flows. Both first filter long texts, repair and remove HTML markup, and annotate sentence and token boundaries (cf. Figure 6.2). Subsequently, the linguistic data flow detects pronouns, negation, passive voice, and parenthesis, while the entity extraction flow first annotates part-of-speech tags and then drug, gene, and disease names using either dictionary or ML-based tools.

We first evaluated both flows on the 20 GB sample with varying DoPs, which led to a number of interesting observations. First, we could not execute the entity extraction data flow with a DoP smaller than 4 due to the excessive runtimes of the ML-based taggers (see above). Furthermore, we could not run this flow with DoPs larger than 28 due to the very high memory requirements of the dictionary-based taggers which each require between 6 and 20 GB of main memory per worker thread. Very likely, this is due to the fact that they transform each dictionary entry (i.e., a regular expression) into a the corresponding non-deterministic finite automaton, which usually greatly increases space requirements. However, the nodes we used have only 24 GB main memory; thus, we could not run more than one instance of these tools per node in the cluster. In contrast, the much less demanding linguistic data flow could be scaled out over the entire range of DoPs without any problems.

As shown in Figure 6.6, scale out for both tested flows was satisfactory until DoP=16 for entity extraction, with a decrease in execution time of up to 72%, and until DoP=12 for the linguistic analysis, with a decrease in execution time of up to 95%. Using more nodes brought only marginal further improvements in execution times. This behavior can be explained by the relatively high start-up times of certain tools. For instance, the dictionary-based gene name recognition algorithm needs approximately 20 minutes (!) to load the dictionary and to create the internal data structures used for text matching. These 20 minutes are a hard lower bound for the runtime of this task, regardless of the number of nodes being used. It is not possible to work around this bound in a non-intrusive manner; one either has to use another tool or perform substantial changes to the tool itself. Scale-out of the linguistic flow was considerably better because in this data flow, startup costs of all involved tasks are negligible.

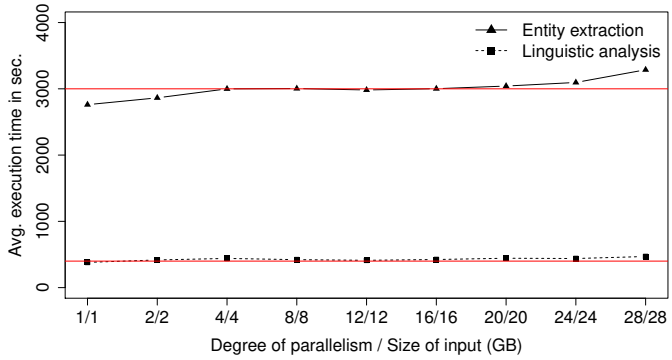


Figure 6.7: Scale up of linguistic and entity extraction partial data flows. Ideal scale up is displayed in red.

Clearly, the concrete DoP beyond which no more performance gains are obtained depends on the size of the input data, which was rather small in our scale-out experiments. Therefore, we also performed scale-up experiments, where we increased the number of available compute nodes synchronously to the amount of input data. As can be seen from Figure 6.7, the linguistic data flow exhibits an almost ideal scale-up, whereas the entity extraction flow scales sub-linear for large DoPs and input sizes, which is consistent to the result of the scale-out experiments.

6.3.3 Processing the entire crawl - a war story.

Although Stratosphere offers an elegant and powerful way of specifying complex IE data flows and is also capable of optimizing and parallelizing them on the given cluster, we could not execute the complete flow on the available hardware. This had mostly three reasons:

First, as described above, the dictionary-based entity taggers come along with very high main memory requirements. The complete data flow as shown in Figure 6.2 needs roughly 60 GB main memory per worker thread, which clearly exceeds the amount of RAM available on each node. Unfortunately, the scheduling component of Stratosphere does not consider memory consumption per worker node as optimization goal, for that reason we could not run an entire flow on any of the nodes.

Second, a severe issue occurred with the ML-based disease recognition tool. This tool brings its own linguistic pre-processing, which is imported from the OpenNLP library, version 1.4[39]. However, all other OpenNLP operators we integrated into Stratosphere (such as tokenization and sentence splitting) are based on version 1.5, which is not downward compatible to 1.4. Unfortunately, the Java classloader employed in the system's runtime engine is not capable of using two different versions of the same library.

Third, in most popular Big Data analysis scenarios, a very large input is scanned and subsequently aggregated into smaller and smaller intermediate results. Accordingly, the bulk of the network traffic for accessing and inputs and writing outputs of tasks is

[39] `http://opennlp.apache.org/` (last accessed: 2016-10-05)

usually spent by those tasks that are at the beginning of the analysis flows, whereas latter tasks often process only small data sets. However, the situation is quite different in a text analytics scenario like ours, where texts are piped through a series of tasks, each adding specific annotations (POS tags, entity annotation, token boundaries etc.) and thus actually increasing the size of the data throughput the analysis pipeline. In our case, the total amount of data produced is 1,6 TB, consisting of 400 GB entity annotations and 1,2 TB linguistic annotations, on top of the 1 TB raw text. Storing and accessing these large intermediate data sets through HDFS over-stressed the cluster network (nodes are connected by a 1 GB switch), leading to unpredictable network delays which in turn led to time-out induced crashes in some of the annotation tools.

To cope with these problems, we had to take several drastic measures. First, we split up the flow into different parts such that each part only required memory within the given limits; essentially, we created one flow for all linguistic analysis and one flow per entity class of the biomedical analysis. Still, the memory requirements (see above) put severe constraints on the number of threads runnable per node, which grossly hampered the overall DoP and thus greatly increased the overall runtime. Eventually, we spinned of gene recognition, being the most space-consuming task, and executed it on a single server with 1 TB RAM using 40 threads. In the same manner, we had to perform disease name extraction in a separate run to overcome versioning problems. To cope with the network problems, we furthermore splitted the crawled data into chunks of 50 GB and executed the different flows separately on these chunks.

Alternatively to splitting the data flow into separate parts and analyzing the crawled data chunk-wise, we considered renting cloud systems for our experiments but quickly disregarded this option due to high rental costs. Since memory requirements of IE operators is the most limiting factor in our study, only cloud instances with more than 35 GB RAM come into question, which are still rather costly. For example, renting a 28 node cluster of instance type "m4.4xlarge" with 64GB of RAM available per node from AWS costs approximately 650 USD per day as of March 2016[40].

6.4 Content analysis

A main interest in this project is to compare extraction results from the web to the scientific literature to identify commonalities and differences between biomedical texts from the web and from peer reviewed journals. Thus, we also analyzed abstracts and full-texts from Medline and PMC using the same IE data flow (downstream from the HTML treatment) as for the crawled data. As a fourth text collection, we used all pages crawled but classified as irrelevant. Table 6.3 summarizes the data sets enclosed in this analysis. In total, we analyzed more than 1 TB of data, whereof 373 GB were web documents identified as relevant to the biomedical domain, 607 GB of irrelevant web documents, and 21 and 67 GB of textual data taken from Medline and Pubmed Central (PMC), respectively. Most documents are contained in Medline with a rather short lengths, whereas the longest documents are contained in the relevant part of the crawl and PMC. The distribution of document lengths is significantly different ($P < 0.01$) in all datasets (cf. Figure 6.8(a)). Web documents were available in HTML format, Medline

[40] `http://aws.amazon.com/ec2/pricing/` (last accessed: 2016-10-05)

	Size in GB	No. of documents	Mean no. of characters
Relevant crawl	373	4,233,523	88,384
Irrelevant crawl	607	17,704,365	37,625
Medline	21	21,686,397	865
PMC	19	250,440	55,704

Table 6.3: Summary of data sets enclosed in corpus quality analysis.

and PMC were given in plain text format, and processed with the data flow shown in Figure 6.2.

We performed an in-depth analysis and comparison of the results obtained on these four corpora. Our analysis is split into a linguistic part, concerning properties such as article lengths, sentence structure, and usage of grammatical structures, and a domain-specific part, comparing the occurrence frequencies and distributions of three biomedical entities, i.e., drugs, genes, and diseases.

6.4.1 Linguistic structure

Analyzing the linguistic structure of texts is important for assessing the complexity of texts and to judge whether existing IE tools, which were trained and developed for different corpora might perform well in web documents. Particularly, we examine

- document and sentence lengths,
- incidence of negation,
- incidence of passives,
- incidence and types of pronouns, and
- incidence and types of parenthesis.

Differences in obtained measures were statistically assessed using the Mann-Whitney-Wilcoxon signed rank test. This test produces a P-value, which estimates the probability that the observed differences are due to random effects in the data.

Document and sentence lengths

Sentence lengths impact IE and NLP in different ways. First, the execution time of IE and NLP tools usually directly depends on the lengths of the sentences to be analyzed. For example, the runtime complexity of automaton-based algorithms performing Named Entity Recognition (NER) using a fixed dictionary of search terms is $O(|search\ terms| + |sentence\ length|)$ [Aho and Corasick, 1975], whereas the time complexity of modern NER methods based on conditional random fields is quadratic in the length of the sentence time [Viterbi, 2006]. Second, the difficulty of constituent and dependency sentence parsing and the difficulty of modern relation extraction methods rises with sentence lengths [Tikk et al., 2013]. Likewise, if crawled web documents contain shorter sentences than Medline or PMC, we expect the former to be easier to analyze. Other

important measures are the document lengths in the different corpora, as these must be considered when comparing the frequency of entity mentions. Sentence length was determined in characters for each sentence in the different data sets.

Figure 6.8(a) displays the distribution of document lengths and Figure 6.8(b) displays mean sentence lengths across the different data sets. Mean document lengths in relevant documents were significantly shorter than in PMC ($P < 0.01$), but significantly longer compared to irrelevant documents ($P < 0.01$) and to Medline abstracts ($P < 0.01$). Document lengths for the relevant corpus show the largest variance, which increases the need for appropriate load balancing in a distributed setting. Differences in sentence lengths between the four corpora are also significant, confirming previous findings from [Cohen et al., 2010] regarding Medline abstracts and PMC full texts. These differences have to be kept in mind when selecting tools for IE that are based on gold standard data. Most tools we are aware of were trained and evaluated on Medline abstracts and thus on rather short sentences; accordingly, we expected a lower performance of these tools on longer sentences than reported in the literature.

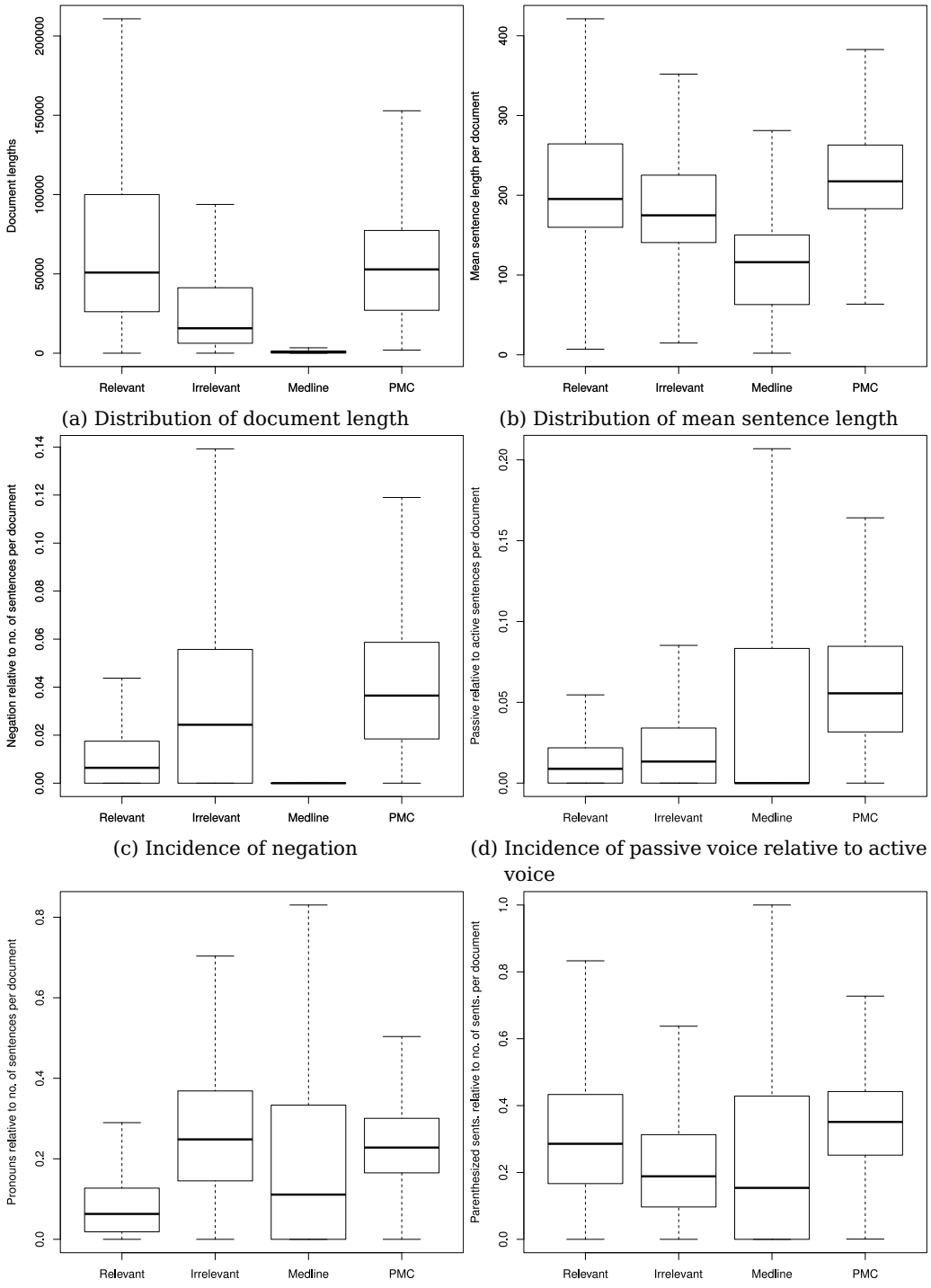
Incidence of negation

Detecting negation is important in many areas of natural language processing (e.g., sentiment analysis, relation extraction) and is particularly important for analyzing biomedical texts [Agarwal and Yu, 2010]. Here, we used a rather simple method for determining negations in sentences, using a set of regular expressions to find mentions of the words *not*, *nor*, and *neither*. As shown in Figure 6.8(c), the incidence of negation in the four corpora is significantly different ($P < 0.01$) regarding the overall incidence of negation and the relative frequency of negation with respect to document length. Specifically, texts in the set of relevant documents have a lower incidence of negation than in PMC and the irrelevant pages and a higher incidence of negation than in Medline. Accordingly, appropriate treatment of negation will be more important for web data than for scientific articles.

Incidence of active and passive voice

Active and passive voice are two different methods of formulating an English sentence that use different types of verbs. Although sentences formulated in active or passive voice have the same semantic meaning, word orders are often changed in passive sentences and different types of verbs are used that pose challenges to syntactic parsers. Specifically, passive verb phrases are often mislabelled as active verb phrases when the auxiliary verb is missing [Igo and Riloff, 2008]. We extracted passive voice from each set of documents using regular expressions searching for the string "ed by". Note that this underestimates the incidence of passive voice (e.g., when agents are missing), but since we applied this method to each data set the comparison is still valid.

As shown in Figure 6.8(d), the incidence of passives is significantly different across the different data sets ($P < 0.01$). The highest mean incidence of passives was found in PMC and the lowest in articles from Medline. Incidence of passives in both relevant and irrelevant documents was comparatively small, indicating that parser errors due to a faulty recognition of passive voice might occur less often than in analyses using the PMC data set.



(a) Distribution of document length

(b) Distribution of mean sentence length

(c) Incidence of negation

(d) Incidence of passive voice relative to active voice

(e) Incidence of pronouns

(f) Incidence of parenthesis

Figure 6.8: Distribution and incidence of linguistic properties per document in different data sets.

Incidence of pronouns

Pronominal anaphora are important in biomedical IE to perform co-reference resolution [Gasperin and Briscoe, 2008]. To measure the amount of such co-references in our corpora, we counted six different classes of pronouns in each data set. Interestingly, the incidence of demonstrative, relative, and object pronouns, which are the most important pronoun classes for co-reference resolution, was significantly lower both in relevant and irrelevant texts compared to texts from PMC (cf. Figure 6.8(e), a distinction between pronoun classes can be found in Appendix 4). We expected this observation concerning irrelevant texts since these texts are significantly shorter than texts from PMC. Our observation is surprising for relevant texts as these are significantly longer than documents from PMC. This finding might indicate that co-reference resolution on crawled texts is not as vital as in analyzing biomedical full-text literature.

Incidence of parentheses

Parentheses can hint to abbreviations, paper references, synonyms of named entities, etc., which are very important during NLP processing. Properly treating parenthesis is also highly important for syntactic parsing, as text in parentheses does typically not conform to the sentence grammar. We extracted parenthesized text using a set of regular expressions and found that that their incidences differ significantly ($P < 0.01$) between all data sets (cf. Figure 6.8(f)). We observed the highest incidence in texts from PMC, followed by relevant web documents and Medline, and the lowest in irrelevant documents.

6.4.2 Corpus quality

The result we were most interested in this study from an application point-of-view was the degree of differences in the biomedical entities that are mentioned on the web versus the scientific literature[41]. Table 6.4 lists the number of distinct names that were found in the two crawled corpora, in Medline, and in PMC for the entity classes disease, drug, and gene distinguished by annotation method. As expected, ML-based annotation produces substantially more annotations for each entity type than dictionary-based annotation. We also notice that the total number of distinct annotations is significantly different between relevant and irrelevant pages both for dictionary- and ML-based approaches, with much larger numbers of annotations in relevant documents for each entity class, which is reassuring of the crawl quality. In the following, we compare named entity annotations in the different corpora more deeply for each entity type.

Disease names

As shown in Figure 6.9(a), the incidence of disease names per document is higher in PMC articles and relevant web documents compared to Medline abstracts and irrelevant crawled documents, which rarely mention more than one disease (presumably mostly false positives with abbreviations). Differences of the mean number of

[41] The analysis of corpus quality in terms of biomedical relevance was carried out jointly with the domain expert Yvonne Lichtblau.

Data set	Annotation Method	Disease	Drug	Gene
Relevant	Dictionary	26,344	17,974	73,435
	Machine learning	629,384	28,660	5,506,579
Irrelevant	Dictionary	5,318	8,456	22,131
	Machine learning	119,638	15,875	991,010
Medline	Dictionary	11,194	12,164	29,928
	Machine learning	343,184	20,282	4,715,194
PMC	Dictionary	12,291	15,013	92,319
	Machine learning	277,211	25,462	1,858,709

Table 6.4: Number of distinct entity names by corpus.

disease annotations per 1000 sentences between relevant ($avg_{rel} = 128.49$) and irrelevant ($avg_{irrel} = 4.57$), relevant and Medline ($avg_{medl} = 204.92$), and Medline and PMC ($avg_{pmc} = 117.51$) are all highly significant ($P < 0.01$). One explanation for the smaller incidence of disease names in Medline abstracts compared to web and PMC full-text documents even when normalized to per-sentence measures is the shorter average sentence length in the abstracts. Furthermore, the large number of disease names in the relevant web document is certainly a result of the way we generated seeds, often using disease names as keywords. Accordingly, our crawl should be enriched for disease-related websites, such as information websites for patients or disease-specific support groups.

Drug names

Drug names are highly heterogeneous, since there exist three different naming conventions: the chemical name (i.e., mostly following the IUPAC nomenclature for chemicals), the generic or non-proprietary name, and brand names. For example, Ibuprofen, a drug used to treat pain and inflammation, has a synonymous chemical name "(RS)-2-(4-(2-methylpropyl)phenyl)propanoic acid" and it is also known under different brand names (e.g., Motrin, Nurofen, Caldolor).

Figure 6.9(b) displays the incidence of drug names per document in the different data sets. In relevant documents, more drug mentions were recognized on average compared to irrelevant documents and abstracts taken from Medline. The means of drug name annotations for both annotation methods combined per 1000 sentences differ significantly ($P < 0.01$) between relevant ($avg_{rel} = 97.83$) and irrelevant ($avg_{irrel} = 6.85$) documents, and between relevant documents and Medline ($avg_{medl} = 293.95$) abstracts and PMC documents ($avg_{pmc} = 275.95$). This again indicates a reasonable discriminative power of the crawling classifier for relevant/irrelevant documents. Possible reasons for the differences between crawled relevant documents and Medline abstracts are the same as for disease names, as disease-related web sites often are also full of drug names. We also found that the means between PMC and Medline differ significantly, which confirms a result from Cohen et al. [Cohen et al., 2010], and which can also be attributed to different document lengths.

Gene names

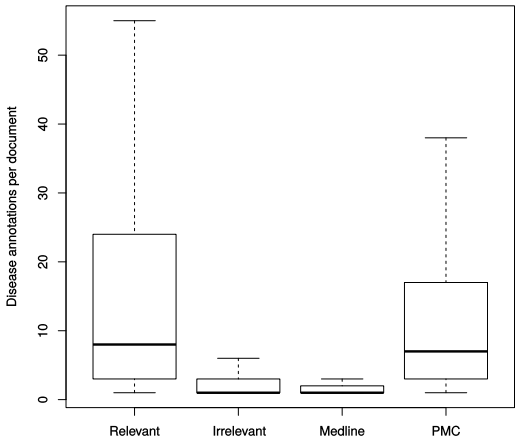
Gene name recognition is considered as one of the most challenging tasks in biomedical named entity recognition, since both the number of existing genes and the variability of gene names is high [Leser and Hakenberg, 2005]. Although naming guidelines for gene names are available for some species, texts mentioning gene names often do not adhere to these rules and many naming variations exist.

Similar to results for disease and drug names, differences in the number of gene mentions per document are significant between relevant and irrelevant documents, and between relevant and Medline ($P < 0.01$) and the means of gene name annotations per 1000 sentences for dictionary-based annotation differ significantly between all sets ($avg_{rel} = 128.23$, $avg_{irrel} = 4.39$, $avg_{medl} = 415.58$, $avg_{pmc} = 74.12$). However, the most striking observation regarding the incidences of gene names are the extremely large differences between the dictionary-based extraction method and the ML-based algorithm. Using ML, we recognized more than 5.5 million distinct gene names in relevant documents, whereas dictionary-based annotation only finds 73,435 different gene names (cf. Table 6.4). It is immediately clear that the vast majority of annotations produced by BANNER must be false positives, because there exist only roughly 900,000 distinct gene names (including synonyms) in the public gene-related databases. The reason for this presumably excessive amount of wrong annotations is the fact that BANNER was trained on a small set of selected Medline abstracts, which exhibit different language characteristics than web documents (see Section 6.4.1). Upon manual inspection, we noticed that a very large number of false positives are three letter acronyms (TLA), which are almost always tagged as genes by our tool; this strategy is correct for the gold standard abstracts used for developing and evaluating the tool, but leads to catastrophic performance on any other documents. Therefore, we filtered all TLAs from the list of ML-tagged gene names prior to further analysis, reducing, for instance, the number of distinct gene names in the relevant web corpus from 5.5 million to 2.3 million. Figure 6.9(c) displays the incidence of gene names in the different data sets after filtering.

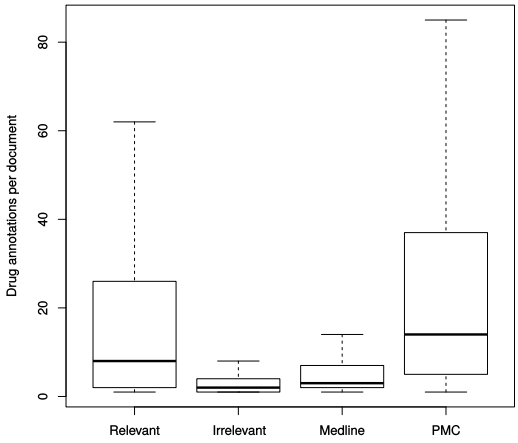
Entity annotation overlap and difference

Finally, we determined the differences in the sets of extracted entities between the different data sets to assess whether focused crawling has the potential to open up new sources of biomedical knowledge and to exclude the potential danger of simply having many scientific abstracts in the relevant crawl. Figure 6.10 shows the overlap and non-overlaps of dictionary-based entity annotations. For all evaluated entity types, we found that the overlap of extracted names between relevant and irrelevant documents is notable but rather small, i.e., approximately 15% for disease names, approximately 30% for drug names (86% out of which were also found in Medline and PMC), and 17% for gene names. The overlap between relevant and Medline and relevant and PMC is considerably larger and ranges from 6% (ML-based gene extraction) to 60% (dictionary-based gene extraction).

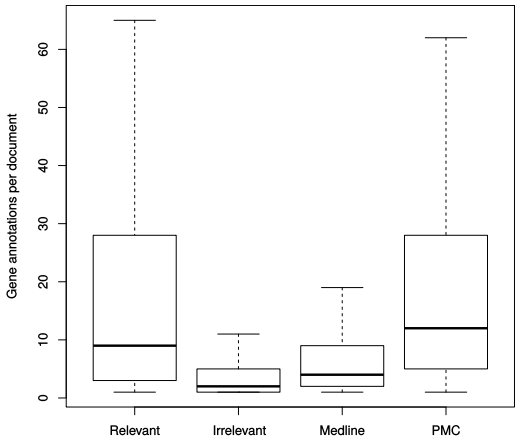
We also assessed the statistical significance of these differences using the Jensen-Shannon divergence (JSD), an information-theoretic measure for assessing the difference between two probability distributions based on the well-known Kullback-Leibler



(a) Diseases



(b) Drugs



(c) Genes

Figure 6.9: Incidence of named entity annotations per document in the different corpora.

Entity	Annotation method	$JSD_{R,I}$	$JSD_{R,M}$	$JSD_{R,P}$	$JSD_{I,M}$	$JSD_{I,P}$	$JSD_{M,P}$
Disease	Dictionary	0.4463	0.3525	0.3354	0.4528	0.3941	0.2978
	Machine learning	0.6954	0.4249	0.4095	0.7181	0.7043	0.3836
Drug	Dictionary	0.5369	0.2864	0.1986	0.5950	0.5535	0.2885
	Machine learning	0.5263	0.2967	0.2026	0.6044	0.5433	0.2759
Gene	Dictionary	0.6548	0.3596	0.1673	0.6850	0.6633	0.3588
	Machine learning	0.7749	0.5103	0.5342	0.7822	0.7741	0.5812

Table 6.5: Jensen-Shannon divergence (JSD) between different corpora with respect to entity types and annotation methods. The letter R in the table heading corresponds to relevant documents, I to irrelevant documents, M to citations taken from Medline, and P to full-texts listed in PMC.

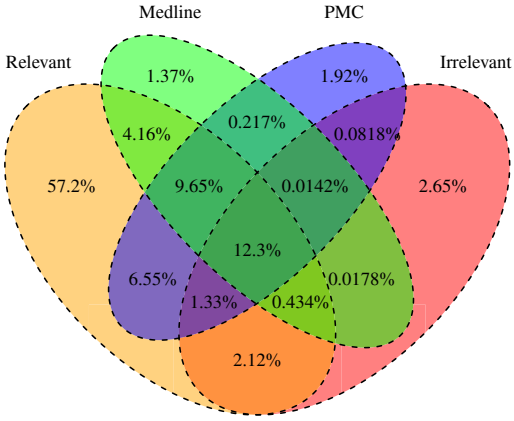
divergence [Lin, 2006]. JSD is a symmetric measure and results in values bounded for two distributions A, B with $0 \leq JSD_{A,B} \leq 1$, where similar distributions approach a JSD close to 0, and dissimilar distributions approach 1.

For each data set and entity type, we determined the probability distribution of entity names and computed for each combination of data sets the JSD (cf. Table 6.5). The comparison of relevant versus irrelevant documents exhibits larger JSDs for any entity type ($JSD_{Rel,Irrel}$ varies between 0.4463 for disease names and 0.6548 for gene names) compared to the JSD of Medline and relevant documents ($JSD_{Rel,Medl}$ varies between 0.2864 for drug names and 0.3596 for gene names), and the JSD of PMC and relevant documents ($JSD_{Rel,PMC}$ varies between 0.1673 for gene names 0.3354 for disease names). Similarly, JSDs between irrelevant documents and Medline ($JSD_{Irrel,Medl}$ varies between 0.4528 for disease names and 0.6850 for gene names) and irrelevant and PMC ($JSD_{Irrel,PMC}$ varies between 0.3941 for disease names and 0.6633 for gene names) are substantially larger. These observations indicate that documents classified as relevant during crawling actually are more similar to the biomedical literature and thus relevant for the target domain.

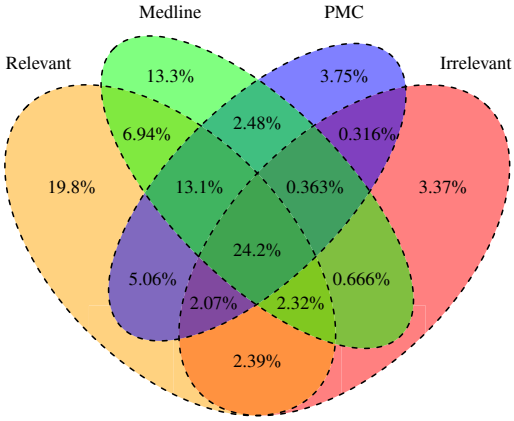
Together, these findings give clear evidence that there is a significant amount of information on the web which is not contained in the scientific literature, indicated by several thousand distinct entity names for each entity type which appear only in relevant web documents.

6.5 Summary and open questions

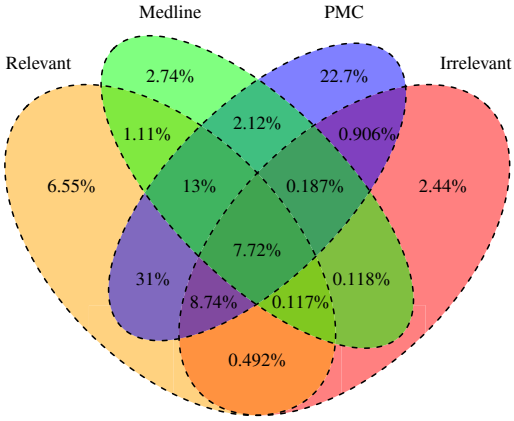
In this chapter, we reported our experiences with a study in crawling and deeply analyzing a large domain-specific corpus from the web, exemplified for the field of biomedical research. From a domain-knowledge point-of-view, our results indicate that there is indeed a large body of biomedical knowledge on the web that is not present in the scientific literature. Clearly, much more research is necessary to substantiate this hypothesis and to assess the usefulness of this knowledge, which could be, for instance, reports of high quality that were not (yet) published or important text book knowledge



(a) Disease



(b) Drug



(c) Gene

Figure 6.10: Annotation overlap of distinct entity names in % for different entity types and dictionary-based annotation.

that is so established that one cannot find scientific publications discussing it. However, a large fraction presumably are also false positive matches of the taggers or information of dubious quality and reliability.

Our study also brought up a number of open questions and technical pitfalls of focused crawling and large-scale IE on crawled web documens, which we now briefly summarize:

Reliable MIME-type detection

Large files downloaded during crawl are often not textual but embedded presentation slides or formatted documents, which were wrongly classified as plain textual. Filtering by document size only, as we did, is not rewarding, since it easily misses relevant (and extensive) content as posted in blogs, personal websites, or on Arxiv.org. However, we are not aware of any robust tools or ongoing research for reliable MIME-type detection; instead, detecting MIME-types usually is carried out by regular expression matching on the file name extension or by analyzing the first n bytes of a document. We used the Apache Tika[42] library during crawling, which ships only with a list of a handful common MIME-types. Although this list can be extended manually with custom types, a manual extension of this list is hardly feasible for web-scale crawling due to the heterogeneity of data available online.

Robust HTML boilerplate detection

According to Ofuonye et al. [2010], 95% of HTML documents on the web do not adhere to W3C HTML standards, most errors can be attributed to missing characters in the markup set (e.g., "<" or ">"), missing information on character set and document type. 13% of the analyzed websites had so severe issues that they could not be transcoded. However, correctly formatted HTML pages seem to be a prerequisite for most boilerplate detection algorithms. In the course of this study, we evaluated different boilerplate detection algorithms and found them to perform reasonably well on a gold-standard set of 1,906 pages (cf. Appendix 4 for details.). Applying these tools to our crawled documents, however, revealed that they are highly sensitive to markup errors, often resulting in crashes or empty results. As a work-around, we integrated a markup repair operator in the analysis process before applying boilerplate detection, which ensured that 94% of the crawled documents survived the markup removal step. Nevertheless, we believe that developing boilerplate detection algorithms that are more robust against errors in real life web pages is essential for seamless and comprehensive text analytics from web documents.

NLP and IE models for web documents

It is well known that ML tools work best on data sets that exhibit similar language characteristics as those used for training. Most research into NER tools in biomedicine is performed on Medline abstracts, both with respect to training data and evaluation data. On such data, ML-based NER is clearly superior to other approaches, as shown in many recent studies and international competitions [Segura-Bedmar et al., 2014].

[42] `http://tika.apache.org` (last accessed: 2016-10-05)

Accordingly, all ML-based methods used in this project employ models trained on Medline abstracts since no other training data is available. However, our study reveals that web documents and documents from Medline and PMC are significantly different in several aspects. This leads to an enormous amount of false positive matches by these tools, which are often short abbreviations. We believe that there is a great need for more sophisticated models for domain-specific entity recognition from web documents. Note that the current research into this direction typically targets rather simple entity types, such as persons, places, or products [Etzioni et al., 2005]. To our knowledge, the performance of such methods on the more difficult biomedical entity types has not yet been evaluated.

Trade-off between precision and yield in focused crawling

When setting-up our system, we focused on a high-precision text classifier as we believed that the number of true positives can be improved more easily with longer crawls than with a high-recall classifier, which might also retrieve many false positive pages. However, we actually observed that this strategy was not as effective as we thought. Actually, the size of the crawl we obtained was bound by the fact that our crawl frontier eventually emptied. As described in Section 6.1, we already had to significantly extend our seed list to obtain a crawl of at least the size we have now. Several strategies could be followed to create larger focused crawl. For instance, one could produce even larger seed lists, but this requires substantial preparation time given the current limits of the search engine APIs. Another approach would be to also follow links from pages classified as irrelevant, but only with a small margin. Finally, one could tune the classifier towards more recall during crawling, and classify each crawled text later a second time with a model geared towards high precision. Which of these ways is the most promising one, remains an open question.

Crawling and text analytics as a consolidated process

This project pursued a two-staged approach, where crawling and text analytics was performed in two separate phases using very different infrastructures. However, the result of the IE pipeline could actually be a valuable input for the classifier during a crawl, as the occurrence of gene names or disease names are strong indicators for biomedical content. We believe it would be a worthwhile undertaking to research systems that would allow specifying crawling strategies, classification, and domain-specific IE in a single system. Such a system would not only greatly reduce the time needed to build web-scale domain-specific text analysis systems, but also has the potential to greatly improve crawl quality since results obtained during entity extraction could be used for proper document classification and thus further improve the focus of a topical crawler.

7 Summary and outlook

7.1 Summary

In this thesis, we presented and evaluated a query-based IE system, which enables scalable and declarative information extraction on the parallel data analytics system Stratosphere. Our system is configurable towards concrete application domains and scalable to large-scale text processing. It enables end-users to formulate complex IE tasks as queries in the structured and declarative language Meteor, which are compiled into logical Sopremo data flows. These data flows are logically optimized with SOFA, translated into parallel data flow programs, and executed on parallel compute infrastructures.

Chapter 2 introduced fundamental terminology, a summary of typical IE tasks, a discussion of existing approaches and systems for IE at large scale fundamental for the remainder of this thesis. We also introduced the parallel data analytics system Stratosphere, its layered architecture, and the query and data flow compilation process in this system focussing on the Meteor query language and the algebraic layer Sopremo.

Chapter 3 introduced domain-independent, algebraic operators addressing all fundamental tasks in information extraction (IE) and web analytics (WA). We showed how end-users can properly combine IE and WA operators in a declarative way to create complex data flows in Meteor for domain-specific applications using a variety of concrete operator instantiations. Furthermore, we showed how elementary operators can be composed into complex operators to ease the definition of complex analytical tasks. Finally, we discussed differences between concrete operator instantiations regarding physical and algebraic properties and pinpointed differences in runtime and startup behaviour to highlight both the potential and importance of optimizing the execution order of data flows with UDFs.

Chapter 4 surveyed practical techniques and the state-of-the-art in optimizing data flows with UDFs. We discussed techniques for syntactical data flow modification, approaches for inferring semantics and rewrite options for UDFs, and methods for data flow transformations both on the logical and on the physical level. This chapter concluded with an overview on declarative data flow languages for parallel data analytics systems from the perspective of their build-in optimization techniques. We found that some of the discussed techniques are available in running systems, although comprehensive optimization of UDFs and non-relational operators still is a true challenge for many systems.

Chapter 5 introduced SOFA, a novel approach for extensible and semantics-aware optimization of data flows with UDFs. SOFA builds upon a concise set of properties for describing the UDF's semantics and it combines automated analysis of UDFs with manual annotations to enable comprehensive data flow optimization. A unique characteristic

of SOFA is extensibility: operators and their properties are arranged into taxonomies, which considerably eases integration and optimization of new operators. We evaluated SOFA on a diverse set of UDF-heavy data flows and compared its performance to three other approaches for data flow optimization. Our experiments revealed that SOFA is able to reorder acyclic data flows from different application domains, leading to considerable runtime improvements. We also showed that SOFA finds plans that outperform plans found by other techniques. Furthermore, we described how SOFA is integrated into the Stratosphere system to enable the end-to-end development, optimization, and execution of data flows that contain UDFs.

Chapter 6 presented a case study, which investigated the real-life applicability of our operator design, extensions to the Meteor query language, and optimization approach in a challenging setting to compare the "web view" on health-related topics with that derived from a controlled scientific corpus. We combined a focused crawler, which applies shallow text analysis and classification to maintain focus, with our text analytics system built inside Stratosphere. All text and web analytics was carried out using a small set of declarative data flows and we systematically evaluated scalability, quality, and robustness of the employed methods and tools. Finally, we summarized lessons learnt during this project and pinpointed the most critical challenges in building an end-to-end IE system for large-scale analytics.

7.2 Outlook

The results of this thesis point to different future research directions. In the following, we summarize the most urgent challenges to further improve scalability of complex IE in parallel data analytics systems to very large text collections:

Holistic data flow optimization

We believe that the parallel processing of general data flows with UDFs would greatly benefit from advancements of the optimization process itself. All modern data flow systems approach optimization multi-staged, where data flows are first simplified, possibly reordered logically and finally optimized physically for parallel execution. This may lead to sub-optimal execution strategies since none of these stages can access and exploit all relevant information for optimization. For example, operator decomposition is useful to enlarge options for logical operator reordering, but possibly prevents operator chaining on the on physical level. We envision a holistic optimization approach, where all relevant information on UDF semantics, rewrite and simplification rules as well as information on physical data properties and the underlying compute infrastructure is used in a single optimization phase to determine an efficient execution strategy. However, since such holistic optimization approaches involve many options for an optimizer to investigate, efficient plan enumeration strategies and cost models are also greatly needed. Unlike relational queries, which mostly translate to linear or tree-shaped execution plans, data flows often exhibit DAG-shaped logical and physical plans. How to efficiently enumerate alternative execution strategies for such plans beyond exhaustive enumeration together with branch-and-bound based pruning still is an open and chal-

lenging question, especially because exhaustive enumeration is NP-hard [Ibaraki and Kameda, 1984].

Benchmarking data flows with UDFs

In Chapter 4, we surveyed a multitude of promising methods for optimizing data flows with UDFs, each addressing different aspects of the optimization process. However, the direct benefits of the presented methods alone and in combination in a concrete system have not been evaluated systematically yet for two reasons. First, we are not aware of any optimizer for a parallel data analytics system implementing all or a majority of the discussed methods, instead, existing systems often rely on data flow optimization based on heuristics or manual data flow transformations performed by the developer. Second, we are not aware of any benchmark in the area of large-scale data processing, which focuses on optimizing UDFs. Quite a few benchmarks for Big Data analytics have been designed in the past years, but these focus mostly on SQL-style processing of mainly structured data [Amplab, 2014; Ghazal et al., 2013], or graph processing [Barnawi et al., 2014; Batarfi et al., 2015; Han et al., 2014]. The most comprehensive benchmark to date is BigBench [Ghazal et al., 2013], which also includes a few data flows with UDFs executed on semi- or unstructured data, but focuses mainly on analyzing structured data. We believe that establishing a benchmark for data flows with UDFs could be very helpful to gain deep insights into the benefits the given a heterogeneous workload containing many UDFs. The queries we developed for evaluating SOFA could be a valuable starting point for such a benchmark (cf. Appendix 3).

Cost estimation of UDFs

In Chapter 5, we introduced a semantics-aware logical optimizer for data flows with UDFs, which enumerates plan alternatives for a given data flow and selects one of those alternatives based on estimated operator and plan costs. Cost estimates therein are based on a model, which combines linear costs for processing the input with operator-specific costs for loading resources necessary for executing UDFs (e.g., loading of dictionaries, models, or indexes). All estimates were determined based on a sample of randomly chosen unstructured documents processed with our operators. We are aware that our estimations are rather rough and may lead the optimizer to select non-optimal plans. We believe that cost estimation would be much more accurate based on workload-specific estimations, for which one could employ statistics collected during data flow execution. How to efficiently retrieve and store such statistics in a distributed setting and how to project the retrieved statistics to new and differently composed data flows is an open question.

Optimization for different extraction goals

Next to throughput, for which we optimize in this thesis, optimization of IE data flows could also target extraction quality as an optimization goal. In this setting, quality would be measured in terms of precision and recall, which are conflicting extraction goals because an increase in recall in IE systems is most often connected to a decrease in precision. Optimization of extraction quality is a very challenging task, since it is not

clear how to accurately model this goal for complex plans and how to balance between precision and recall. For example, when optimizing for high recall, this not necessarily means to select only those operator instantiations based on recall, because the combined effect be too strong and yield many irrelevant results. Moreover, the assumption of independence of errors is often wrong in complex IE tasks, since many operators tend to make mistakes on the same type of input. For example, sentence splitting and part of speech tagging both tend to produce errors on very long sentences. How to accurately reflect such observations in a quality-aware cost model is another open and challenging research question.

Memory-aware scheduling

Even though the NLP and IE tools we used require only a moderate amount of memory for each running instance, these numbers sum-up notably when combining them to complex extraction and analysis data flows and when running multiple instances on a multi-threaded machine. In our case study presented in Chapter 6, this grossly hampered the degree of parallelism we could achieve, leading to sub-optimal resource usage and long analysis times. Furthermore, several tools produced Java out-of-memory errors when applied to long texts. Moreover, dictionary-based entity annotation using very large dictionaries caused severe problems and occasionally crashes, since memory consumption increased dramatically. This observation is directly related to the size of the dictionary, since splitting the dictionary into smaller parts and successively running the gene name annotation operator prevented this behaviour. Therefore, we argue that more research for memory-aware scheduling would greatly help to increase error-resiliency in modern parallel data analytics systems.

I/O efficiency in parallel data analytics systems for large-scale IE

Another important, yet mostly unaddressed research direction is improving the I/O efficiency of parallel data analytics systems for write-intensive applications, such as large-scale IE. Different from other Big Data applications, where huge input data sets are aggregated and reduced to a few Gigabytes, intermediate and final result sets in text analytics grow large and easily exceed the size of the input due to the complex text annotation process. For example, the result sets of all entity annotations produced in our study presented in Chapter 6 reached a size of 400 GB taken together and linguistic annotations comprised around 1,2 TB, which exceeds the size of the input by 60% and creates great challenges for downstream statistical analyses. Temporary intermediate files, which are created during record serialization, can reach sizes of hundreds of Gigabytes and cause hard disks to fill up. Compression of intermediate and final data is rewarding to reduce the amount of data to be written and recent research has shown that adaptive compression depending on the workload in Map/Reduce systems is promising to increase throughput for Big Data applications [Chen et al., 2010; Zou et al., 2014].

Bibliography

S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded DSL for high performance big data processing. In *Proc. Int. Workshop on End-to-end Management of Big Data (BigData '12), held in conjunction with VLDB '12*, pages 1–10, 2012.

F. N. Afrati, D. Delorey, M. Pasumansky, and J. D. Ullman. Storing and querying tree-structured records in Dremel. *Proc. VLDB Endow.*, 7(12):1131–1142, 2014.

S. Agarwal and H. Yu. Biomedical negation scope detection with conditional random fields. *J. Am. Med. Inform. Assn.*, 17(6):696–701, 2010.

E. Agichtein and L. Gravano. QXtract: A Building Block for Efficient Information Extraction from Text Databases. In *Proc. 2003 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '03)*, pages 663–663, 2003.

E. Agichtein and S. Sarawagi. Scalable information extraction and integration. *Tutorial. 12th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2006.

A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, 1975.

A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere Platform for Big Data Analytics. *VLDB J.*, 23(6):939–964, 2014.

A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. Implicit Parallelism Through Deep Language Embedding. In *Proc. 2015 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '15)*, pages 47–61, 2015.

W. Y. Alkowaileet, S. Alsubaiee, M. J. Carey, T. Westmann, and Y. Bu. Large-scale Complex Analytics on Semi-structured Datasets Using asterixDB and Spark. *Proc. VLDB Endow.*, 9(13):1585–1588, 2016.

S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, 2014.

Amplab. Big Data Benchmark, 2014. URL `https://amplab.cs.berkeley.edu/benchmark/`.

P. M. G. Apers, A. R. Hevner, and S. B. Yao. Optimization Algorithms for Distributed Queries. *IEEE Trans. Software Eng.*, 9(1):57–68, 1983.

M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proc. 2015 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '15)*, pages 1383–1394, 2015.

A. Auger and C. Barrière. Pattern-based approaches to semantic relation extraction: A state-of-the-art. *Terminology*, 14(1):1–19, 2008.

F. Bajaber, R. Elshawi, O. Batarfi, A. Altalhi, A. Barnawi, and S. Sakr. Big Data 2.0 Processing Systems: Taxonomy and Open Challenges. *J. Grid Computing*, 14(3):379–405, 2016.

J. Baldridge. The Apache OpenNLP project, 2005. URL `http://opennlp.apache.org`.

B. Baldwin and B. Carpenter. LingPipe, 2003. URL `http://alias-i.com/lingpipe`.

M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open Information Extraction from the Web. In *Proc. 20th Int. Joint Conf. on Artifical Intelligence (IJCAI '07)*, pages 2670–2676, 2007.

A. Barnawi, O. Batarfi, S. Beheshti, R. E. Shawi, A. G. Fayoumi, R. Nouri, and S. Sakr. On Characterizing the Performance of Distributed Graph Computation Platforms. In *Proc. 6th TPC Technology Conference (TPCTC '14) - Performance Characterization and Benchmarking. Traditional to Big Data.*, pages 29–43, 2014.

O. Batarfi, R. E. Shawi, A. G. Fayoumi, R. Nouri, S.-M.-R. Beheshti, A. Barnawi, and S. Sakr. Large Scale Graph Processing Systems: Survey and an Experimental Evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.

D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In *Proc. 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 119–130, 2010.

P. A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM J. Comput.*, 10(4): 751–771, 1981.

P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. R. Jr. Query Processing in a System for Distributed Databases (SDD-1). *ACM Trans. Database Syst.*, 6(4):602–625, 1981.

G. B. Berriman and S. L. Groom. How Will Astronomy Archives Survive the Data Tsunami? *Commun. ACM*, 54(12):52–56, 2011.

K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *Proc. VLDB Endow.*, 4(12):1272–1283, 2011.

S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MaPreduce. In *Proc. 2010 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '10)*, pages 975–986, 2010.

B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-intensive Computing. In *Proc. 27th IEEE Int. Conf. on Data Engineering (ICDE '11)*, pages 1151–1162, 2011.

V. R. Borkar, Y. Bu, E. P. C. Jr., N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: a data model-agnostic compiler backend for big data languages. In *Proc. 6th ACM Symposium on Cloud Computing (SoCC '15)*, pages 422–433, 2015.

D. Borthakur. HDFS architecture guide, 2008. URL `http://hadoop.apache.org/common/docs/current/hdfsdesign.pdf`.

T. Bray. The javascript object notation (json) data interchange format, 2014. URL `http://tools.ietf.org/html/rfc7159`.

A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph Structure in the Web. *Comput. Netw.*, 33(1-6):309–320, 2000.

A. P. G. Brown. Optimization of the order in which the comparisons of the components of a boolean query expression are applied to a database record stored as a byte stream. Patent, US 5794227 A, United States of America, 1998.

J. Burge, K. Munagala, and U. Srivastava. Ordering pipelined query operators with precedence constraints. Technical report, Stanford University, 2005.

M. J. Cafarella and O. Etzioni. A search engine for natural language applications. In *Proc. 14th Int. Conf. on World Wide Web (WWW '05)*, pages 442–452, 2005.

M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *Proc. 13th Int. Workshop on the Web and Databases (WebDB '10)*, pages 10:1–10:6, 2010.

M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. WebTables: Exploring the Power of Tables on the Web. *Proc. VLDB Endow.*, 1(1):538–549, 2008.

P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 4 (38):28–38, 2015.

E. P. Carman Jr., T. Westmann, V. R. Borkar, M. J. Carey, and V. J. Tsotras. A Scalable Parallel XQuery Processor. In *Proc. IEEE Int. Conf. on Big Data (Big Data '15)*, pages 164–173, 2015.

J. Carroll, T. Briscoe, and A. Sanfilippo. Parser evaluation: a survey and a new proposal. In *Proc. 1st Int. Conf. on Language Resources and Evaluation (LREC '98)*, pages 447–454, 1998.

R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

S. Chakrabarti, M. van den Berg, and B. Dom. Focused Crawling: A New Approach to Topic-specific Web Resource Discovery. *Comput. Netw.*, 31(11-16):1623–1640, 1999.

C. Y. Chan and Y. E. Ioannidis. Bitmap Index Design and Evaluation. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '98)*, pages 355–366, 1998.

Y. S. Chan and D. Roth. Exploiting Background Knowledge for Relation Extraction. In *Proc. 23rd Int. Conf. on Computational Linguistics (COLING '10)*, pages 152–160, 2010.

B. Chandramouli, J. Goldstein, and S. Duan. Temporal Analytics on Big Data for Web Advertising. In *Proc. 28th IEEE Int. Conf. on Data Engineering (ICDE '12)*, pages 90–101, 2012.

J.-P. Chanod and P. Tapanainen. Tagging French: comparing a statistical and a constraint-based method. In *Proc. 7th Conf. of European Chapter of the Association for Computational Linguistics (EACL '95)*, pages 149–156. Morgan Kaufmann Publishers Inc., 1995.

B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing:ASQL Implementation On The MapReduce Framework. *Proc. VLDB Endow.*, 4(12):1318–1327, 2011.

S. Chaudhuri and K. Shim. Optimization of Queries with User-Defined Predicates. *ACM Trans. Database Syst.*, 24(2):177–228, 1999.

S. Chaudhuri, U. Dayal, and V. Narasayya. An Overview of Business Intelligence Technology. *Commun. ACM*, 54(8):88–98, 2011.

F. Chen, A. Doan, J. Yang, and R. Ramakrishnan. Efficient Information Extraction over Evolving Text Data. In *Proc. 24th IEEE Int. Conf. on Data Engineering (ICDE '08)*, pages 943–952, 2008.

Y. Chen, A. Ganapathi, and R. H. Katz. To Compress or Not to Compress - Compute vs. IO Tradeoffs for Mapreduce Energy Efficiency. In *Proc. 1st ACM SIGCOMM Workshop on Green Networking*, pages 23–28, 2010.

L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. R. Reiss, and S. Vaithyanathan. System T: An Algebraic Approach to Declarative Information Extraction. In *Proc. 48th Annual Meeting of the Association for Computational Linguistics (ACL '10)*, 2010a.

L. Chiticariu, R. Krishnamurthy, Y. Li, F. Reiss, and S. Vaithyanathan. Domain Adaptation of Rule-based Annotators for Named-entity Recognition Tasks. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP '10)*, pages 1002–1012, 2010b.

A. M. Cohen and W. R. Hersh. A survey of current work in biomedical text mining. *Brief. Bioinform.*, 6(1):57–71, 2005.

K. B. Cohen, H. L. Johnson, K. Verspoor, C. Roeder, and L. Hunter. The structural and content aspects of abstracts versus bodies of full text journal articles are different. *BMC Bioinformatics*, 11(1):492, 2010.

L. Covolo, S. Mascaretti, A. Caruana, G. Orizio, L. Caimi, and U. Gelatti. How has the flu virus infected the Web? 2010 influenza and vaccine information available on the Internet. *BMC Public Health*, 13(1):83, 2013.

H. Cunningham. GATE, a General Architecture for Text Engineering. *Computers and the Humanities*, 36(2):223–254, 2002.

A. Cuzzocrea, I.-Y. Song, and K. C. Davis. Analytics over large-scale multidimensional data: the big data revolution! In *Proc. 14th Int. Workshop on Data Warehousing and OLAP (DOLAP '11)*, pages 101–104, 2011.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

B. D. Davison. Topical Locality in the Web. In *Proc. 23rd Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR '00)*, pages 272–279, 2000.

J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th USENIX Symposium on Operating System Design and Implementation (OSDI '04)*, pages 137–150, 2004.

J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *Commun. ACM*, 53(1):72–77, 2010.

J. Ding, D. Berleant, D. Nettleton, and E. Wurtele. Mining MEDLINE: abstracts, sentences, or phrases. In *Proc. Pacific Symposium on Biocomputing (PSB '02)*, volume 7, pages 326–337, 2002.

C. Doulkeridis and K. Nørvåg. A survey of large-scale analytical query processing in MapReduce. *VLDB J.*, 23(3):355–380, 2014.

M. T. Egner, M. Lorch, and E. Biddle. UIMA GRID: Distributed Large-scale Text Analysis. In *Proc. 7th IEEE Int. Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 317–326, 2007.

EMC Digital Universe. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things, 2015. URL `https://www.emc.com/leadership/digital-universe/2014iview/index.htm`.

O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale Information Extraction in Knowitall: (Preliminary Results). In *Proc. 13th Int. Conf. on World Wide Web (WWW '04)*, pages 100–110, 2004.

O. Etzioni, M. Cafarella, D. Downey, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Unsupervised Named-entity Extraction from the Web: An Experimental Study. *Artif. Intell.*, 165(1):91–134, 2005.

O. Etzioni, M. Banko, S. Soderland, and D. S. Weld. Open Information Extraction from the Web. *Commun. ACM*, 51(12):68–74, 2008.

S. Ewen. *Programming abstractions, compilation, and execution techniques for massively parallel data analysis*. Dissertation, Technische Universität Berlin, 2014.

X. Fan, Z. Guo, H. Jin, X. Liao, J. Zhang, H. Zhou, S. McDirmid, W. Lin, J. Zhou, and L. Zhou. Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE. *IEEE Trans. Parall Distr.*, 26(6):1718–1731, 2015.

R. Feldman and J. Sanger. *Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, 2006.

D. Ferrucci and A. Lally. UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. *Nat. Lang. Eng.*, 10(3-4):327–348, 2004.

T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi, C. Schallhart, and C. Wang. DIADEM: Thousands of Websites to a Single Database. *Proc. VLDB Endow.*, 7(14):1845–1856, 2014.

R. A. Ganski and H. K. T. Wong. Optimization of Nested SQL Queries Revisited. *SIGMOD Rec.*, 16(3):23–33, 1987.

C. Gasperin and T. Briscoe. Statistical Anaphora Resolution in Biomedical Texts. In *Proc. 22nd International Conference on Computational Linguistics (COLING '08) - Volume 1*, pages 257–264, 2008.

A. Gates, J. Dai, and T. Nair. Apache Pig's Optimizer. *IEEE Data Eng. Bull.*, 36(1):34–45, 2013.

M. Gerner, G. Nenadic, and C. M. Bergman. LINNAEUS: A species name identification system for biomedical literature. *BMC Bioinformatics*, 11:85, 2010.

A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *Proc. 2013 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '13)*, pages 1197–1208, 2013.

G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

G. Graefe. Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Trans. Data Eng.*, 6(1):120–135, 1994.

G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18 (3):19–29, 1995.

G. Graefe. Parallel Query Execution Algorithms. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 2030–2035. Springer Publishing Company, Incorporated, 2009.

J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

R. Grishman and J. Sterling. Information Extraction and Semantic Constraints. In *Proc. 13th Int. Conf. on Computational Linguistics (COLING '90)*, pages 355–357, 1990.

R. Grishman, S. Huttunen, and R. Yangarber. Information extraction for enhanced access to disease outbreak reports. *J. Biomed. Inform.*, 35(4):236–246, 2002.

T. Güngör. Part-of-Speech Tagging. In *Handbook of Natural Language Processing, Second Edition*. CRC Press, Taylor and Francis Group, 2010.

Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting Code Optimizations in Data-parallel Pipelines Through PeriSCOPE. In *Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 121–133, 2012.

M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. *Proc. VLDB Endow.*, 7(12): 1047–1058, 2014.

M. Z. Hanani. An Optimal Evaluation of Boolean Expressions in an Online Query System. *Commun. ACM*, 20(5):344–347, 1977.

M. Hausenblas and J. Nadeau. Apache drill: interactive ad-hoc analysis at scale. *Big Data*, 1(2):100–104, 2013.

M. A. Hearst. Automatic Acquisition of Hyponyms from Large Text Corpora. In *Proc. 14th Conf. on Computational Linguistics (COLING '92) - Volume 2*, pages 539–545, 1992.

A. Heise. *Data cleansing and integration operators for a parallel data analytics platform*. Dissertation, Universität Potsdam, 2015.

A. Heise and F. Naumann. Integrating open government data with stratosphere for more transparency. *J. Web Semant.*, 14:45–56, 2012.

A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann. Meteor/Sopremo: An Extensible Query Language and Operator Model. In *Proc. Int. Workshop on End-to-end Management of Big Data (BigData '12), held in conjunction with VLDB '12*, pages 1–10, 2012.

J. M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. *ACM Trans. Database Syst.*, 23(2):113–157, 1998.

J. M. Hellerstein and J. F. Naughton. Query Execution Techniques for Caching Expensive Methods. In *Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '96)*, pages 423–434, 1996.

M. A. Hernández and S. J. Stolfo. The Merge/Purge Problem for Large Databases. In *Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '95)*, pages 127–138, 1995.

M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, 2014.

D. Howe, M. Costanzo, P. Fey, T. Gojobori, L. Hannick, W. Hide, D. P. Hill, R. Kania, M. Schaeffer, S. St Pierre, et al. Big data: The future of biocuration. *Nature*, 455 (7209):47–50, 2008.

F. Hueske. *Specification and Optimization of Analytical Data Flows*. Dissertation, Technische Universität Berlin, 2015.

F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the Black Boxes in Data Flow Optimization. *Proc. VLDB Endow.*, 5(11):1256–1267, 2012.

F. Hueske, A. Krettek, and K. Tzoumas. Enabling Operator Reordering in Data Flow Programs Through Static Code Analysis. *CoRR*, abs/1301.4200:1–4, 2013.

T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

S. Igo and E. Riloff. Learning to Identify Reduced Passive Verb Phrases with a Shallow Parser. In *Proc. 23rd National Conference on Artificial Intelligence (AAAI '08) - Volume 3*, AAAI'08, pages 1458–1461, 2008.

P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. Towards a Query Optimizer for Text-centric Tasks. *ACM Trans. Database Syst.*, 32(4), 2007.

O. Irsoy, O. T. Yildiz, and E. Alpaydin. Design and Analysis of Classifier Learning Experiments in Bioinformatics: Survey and Case Studies. *IEEE/ACM Trans. Comput. Biol. Bioinf.*, 9(6):1663–1675, 2012.

M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *Proc. 2009 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '09)*, pages 987–994, 2009.

A. Jacobs. The Pathologies of Big Data. *Commun. ACM*, 52(8):36–44, 2009.

E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. *Proc. VLDB Endow.*, 4(6):385–396, 2011.

Y. Kano, R. Dorado, L. McCrohon, S. Ananiadou, and J. Tsujii. U-Compare: An Integrated Language Resource Evaluation Platform Including a Comprehensive UIMA Resource Library. In *Proc. 7th Int. Conf. on Language Resources and Evaluation (LREC '10)*, 2010.

V. N. Khuc, C. Shivade, R. Ramnath, and J. Ramanathan. Towards Building Large-scale Distributed Systems for Twitter Sentiment Analysis. In *Proc. 27th Annual ACM Symposium on Applied Computing (SIGAPP '12)*, pages 459–464, 2012.

W. Kim. On Optimizing an SQL-like Nested Query. *ACM Trans. Database Syst.*, 7(3): 443–469, 1982.

J. M. Kleinberg. Hubs, Authorities, and Communities. *ACM Comput. Surv.*, 31(4es), 1999.

C. Kohlschütter, P. Fankhauser, and W. Nejdl. Boilerplate Detection Using Shallow Text Features. In *Proc. 3rd ACM Int. Conf. on Web Search and Data Mining (WSDM '10)*, pages 441–450, 2010.

M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proc. 7th Biennial Conf. on Innovative Data Systems (CIDR '15)*, pages 1–10, 2015.

G. Kougka and A. Gounaris. Declarative Expression and Optimization of Data-Intensive Flows. In *Proc. 15th Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK '13)*, pages 13–25, 2013.

M. Krallinger, F. Leitner, O. Rabal, M. Vazquez, J. Oyarzabal, and A. Valencia. Overview of the chemical compound and drug name recognition (CHEMDNER) task. In *BioCreative Challenge Evaluation Workshop*, volume 2, pages 2–33, 2013.

S. Krause. *Relation Extraction with Massive Seed and Large Corpora*. Diploma thesis, Humboldt-Universität zu Berlin, 2012.

R. Krovetz. Viewing Morphology As an Inference Process. In *Proc. 16th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR '93)*, pages 191–202, 1993.

R. Leaman and G. Gonzalez. BANNER: An Executable Survey of Advances in Biomedical Named Entity Recognition. In *Proc. Pacific Symposium on Biocomputing (PSB '08)*, pages 652–663, 2008.

K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel Data Processing with MapReduce: A Survey. *SIGMOD Rec.*, 40(4):11–20, 2012.

U. Leser and J. Hakenberg. What Makes a Gene Name? Named Entity Recognition in the Biomedical Literature. *Brief. Bioinform.*, 6(4):357–369, 2005.

F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using MapReduce. *ACM Comput. Surv.*, 46(3):31:1–31:42, 2014.

Z. Li. Parsing the Internal Structure of Words: A New Paradigm for Chinese Word Segmentation. In *Proc. 49th Annual Meeting of the Association for Computational Linguistics (ACL '11) - Volume 1*, pages 1405–1414, 2011.

H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *Proc. VLDB Endow.*, 5(11):1196–1207, 2012.

J. Lin. Divergence Measures Based on the Shannon Entropy. *IEEE Trans. Inform. Theory*, 37(1):145–151, 2006.

J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

E. Loper and S. Bird. NLTK: The Natural Language Toolkit. In *Proc. ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, pages 63–70, 2002.

C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

V. Marx. Biology: The big challenges of big data. *Nature*, 498(7453):255–260, 2013.

E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proc. 2006 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '06)*, pages 706–706, 2006.

S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.*, 3(1):330–339, 2010.

D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218(5136):19–22, 1968.

D. Miner and A. Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly Media, Inc., 2012.

J. Minker and R. G. Minker. Optimization of Boolean Expressions-Historical Developments. *IEEE Ann. Hist. Comput.*, 2(3):227–238, 1980.

M.-F. Moens. *Information extraction: algorithms and prospects in a retrieval context*, volume 21. Springer Science & Business Media, 2006.

D. G. Murray, M. Isard, and Y. Yu. Steno: Automatic Optimization of Declarative Queries. In *Proc. 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '11)*, pages 121–131, 2011.

D. Nadeau and S. Sekine. A survey of named entity recognition and classification. *Lingvist. Invest.*, 30(1):3–26, 2007.

T. Neumann. Query Optimization (in Relational Databases). In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 2273–2278. Springer Publishing Company, Incorporated, 2009.

J. Nioche. Behemoth – Large scale document processing with Hadoop, 2012. URL `https://github.com/DigitalPebble/behemoth/`.

J. Nivre. Statistical Parsing. In *Handbook of Natural Language Processing, Second Edition*. CRC Press, Taylor and Francis Group, 2010.

E. Ofuonye, P. Beatty, S. Dick, and J. Miller. Prevalence and classification of web page defects. *Online Inform. Rev.*, 34(1):160–174, 2010.

E. Ogasawara, J. Dias, V. Silva, F. Chirigati, D. Oliveira, F. Porto, P. Valduriez, and M. Mattoso. Chiron: a parallel engine for algebraic scientific workflows. *Concurr. Comput. : Pract. Exper.*, 25(16):2327–2341, 2013.

E. S. Ogasawara, D. de Oliveira, P. Valduriez, J. Dias, F. Porto, and M. Mattoso. An Algebraic Approach for Data-Centric Scientific Workflows. *Proc. VLDB Endow.*, 4(12):1328–1339, 2011.

C. Olston and M. Najork. Web Crawling. *Found. Trends Inf. Retr.*, 4(3):175–246, 2010.

C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proc. 2008 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '08)*, pages 1099–1110, 2008.

M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, Inc., 1999.

L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

C. D. Paice. An Evaluation Method for Stemming Algorithms. In *Proc. 17th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR '94)*, pages 42–50, 1994.

B. Pang and L. Lee. Opinion mining and sentiment analysis. *Found. Trends Inf. Retr.*, 2 (1-2):1–135, 2008.

G. Pant and P. Srinivasan. Learning to Crawl: Comparing Classification Schemes. *ACM T. Inform. Syst.*, 23(4):430–462, 2005.

P. Pantel, D. Ravichandran, and E. H. Hovy. Towards Terascale Semantic Acquisition. In *Proc. 20th Int. Conf. on Computational Linguistics (COLING '04)*, 2004.

R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

M. F. Porter. An algorithm for suffix stripping. In K. Sparck Jones and P. Willett, editors, *Readings in Information Retrieval*, pages 313–316. Morgan Kaufmann Publishers Inc., 1997.

M. S. Raisinghani. *Business Intelligence in the Digital Economy: Opportunities, Limitations and Risks: Opportunities, Limitations and Risks*. Igi Global, 2003.

F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An Algebraic Approach to Rule-Based Information Extraction. In *Proc. 24th IEEE Int. Conf. on Data Engineering (ICDE '08)*, pages 933–942, 2008.

A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. SOFA: An Extensible Logical Optimizer for UDF-heavy Dataflows. *CoRR*, abs/1311.6335, 2013.

A. Rheinländer, M. Beckmann, A. Kunkel, A. Heise, T. Stoltmann, and U. Leser. Versatile Optimization of UDF-heavy Data Flows with Sofa. In *Proc. 2014 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '14)*, pages 685–688, 2014.

A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. SOFA: An extensible logical optimizer for UDF-heavy data flows. *Inf. Syst.*, 52:96–125, 2015.

A. Rheinländer, M. Lehmann, A. Kunkel, J. Meier, and U. Leser. Potential and Pitfalls of Domain-Specific Information Extraction at Web Scale. In *Proc. 2016 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '16)*, pages 759–771, 2016.

A. Rheinländer, U. Leser, and G. Graefe. Optimization of complex dataflows with user-defined functions. *ACM Comput. Surv.*, 50(3):38:1–38:39, 2017.

T. Rocktäschel, M. Weidlich, and U. Leser. ChemSpot: a hybrid system for chemical named entity recognition. *Bioinformatics*, 28(12):1633–1640, 2012.

M. T. Roth and P. M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proc. 23rd Int. Conf. on Very Large Data Bases (VLDB '97)*, pages 266–275, 1997.

G. Rumi, C. Colella, and D. Ardagna. Optimization Techniques within the Hadoop Ecosystem: A Survey. In *Proc. 16th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '14)*, pages 437–444, 2014.

Y. Sagiv. Optimizing datalog programs. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '87)*, pages 349–362, 1987.

S. Sakr, A. Liu, D. Batista, and M. Alomari. A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Commun. Surveys Tuts.*, 13(3):311–336, 2011.

S. Sakr, A. Liu, and A. G. Fayoumi. The family of mapreduce and large-scale data processing systems. *ACM Comput. Surv.*, 46(1):Article No. 11, 2013.

A. L. d. M. Santos. *Compilation by transformation in non-strict functional languages*. Ph.d. thesis, University of Glasgow, 1995.

S. Sarawagi. Information Extraction. *Found. Trends Databases*, 1(3):261–377, 2008.

A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman. Upper and Lower Bounds on the Cost of a Map-reduce Computation. *Proc. VLDB Endow.*, 6(4):277–288, 2013.

H. Schmid. Improvements In Part-of-Speech Tagging With an Application To German. In *Proc. ACL SIGDAT-Workshop*, pages 47–50, 1995.

I. Segura-Bedmar, P. Martínez, and M. Herrero-Zazo. Lessons learnt from the DDIExtraction-2013 Shared Task. *J. Biomed. Inform.*, 51:152–164, 2014.

P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. 1979 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '79)*, pages 23–34, 1979.

W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative Information Extraction Using Datalog with Embedded Extraction Predicates. In *Proc. 33rd Int. Conf. on Very Large Data Bases (VLDB '07)*, VLDB '07, pages 1033–1044. VLDB Endowment, 2007.

Y. N. Silva, P.-A. Larson, and J. Zhou. Exploiting Common Subexpressions for Cloud Query Processing. In *Proc. 28th IEEE Int. Conf. on Data Engineering (ICDE '12)*, pages 1337–1348, 2012.

A. Simitsis, P. Vassiliadis, and T. Sellis. Optimizing ETL Processes in Data Warehouses. In *Proc. 21st Int. Conf. on Data Engineering (ICDE '05)*, pages 564–575, 2005.

A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing Analytic Data Flows for Multiple Execution Engines. In *Proc. 2012 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '12)*, pages 829–840, 2012.

D. Simmen, E. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In *Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '96)*, pages 57–67, 1996.

L. Smith, T. Rindflesch, W. J. Wilbur, et al. MedPost: a part-of-speech tagger for bioMedical text. *Bioinformatics*, 20(14):2320–2321, 2004.

A. Spengler and P. Gallinari. Learning to Extract Content from News Webpages. In *Proc. 23rd Int. Conf. on Advanced Information Networking and Applications Workshops (AINA '09)*, pages 709–714, 2009.

U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query Optimization over Web Services. In *Proc. 32nd Int. Conf. on Very Large Data Bases (VLDB '06)*, pages 355–366, 2006.

K. Stocker, D. Kossmann, R. Braumandl, and A. Kemper. Integrating Semi-Join-Reducers into State of the Art Query Processors. In *Proc. 17th Int. Conf. on Data Engineering (ICDE '01)*, pages 575–584, 2001.

F. Suchanek. Information extraction for ontology learning. In Lehmann and Völker, editors, *Perspectives On Ontology Learning*, pages 135–151. Jens Lehmann and Johanna Völker, 2014.

V. Tablan, I. Roberts, H. Cunningham, and K. Bontcheva. GATECloud. net: a platform for large-scale, open-source text processing on the cloud. *Philos. T. Roy. Soc. A.*, 371 (1983):20120071, 2013.

D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience: Research Articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, 2005.

The Apache Software Foundation. Apache UIMA Asynchronous Scaleout, 2012. URL `https://uima.apache.org/doc-uimaas-what.html`.

The Stanford Natural Language Processing Group. Stanford NLP Software, 2016. URL `http://nlp.stanford.edu/software/`.

P. Thomas, J. Starlinger, A. Vowinkel, S. Arzt, and U. Leser. Geneview: a comprehensive semantic search engine for pubmed. *Nucleic acids research*, 40(W1):W585–W591, 2012.

A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

D. Tikk, I. Solt, P. E. Thomas, and U. Leser. A detailed error analysis of 13 kernel methods for protein-protein interaction extraction. *BMC Bioinformatics*, 14:12, 2013.

K. Tomanek, J. Wermter, and U. Hahn. Sentence and token splitting based on conditional random fields. In *Proc. 10th Conf. Pacific Association for Computational Linguistics (PACLING '07)*, pages 49–57, 2007.

P. Valduriez. Join Indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

A. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Trans. Inform. Theory*, 13(2):260–269, 2006.

H. Wachsmuth, B. Stein, and G. Engels. Constructing Efficient Information Extraction Pipelines. In *Proc. 20th ACM Int. Conf. on Information and Knowledge Management (CIKM '11)*, pages 2237–2240, 2011.

J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical report, Harvard University, 1994.

C. Wang, J. Wang, X. Xie, and W.-Y. Ma. Mining Geographic Knowledge Using Location Aware Topic Model. In *Proc. 4th ACM Workshop on Geographical Information Retrieval (GIR '07)*, pages 65–70, 2007.

D. Warneke. *Massively Parallel Data Processing on Infrastructure as a Service Platforms*. Dissertation, Technische Universität Berlin, 2011.

D. Warneke and O. Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In *Proc. 2nd Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS '09)*, pages 8:1–8:10, 2009.

T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

L. Wörteler, M. Grossniklaus, C. Grün, and M. H. Scholl. Function Inlining in XQuery 3.0 Optimization. In *Proc. 15th Symposium on Database Programming Languages (DBPL '15)*, pages 45–48, 2015.

S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proc. 2nd ACM Symposium on Cloud Computing (SoCC '11)*, pages 12:1–12:13, 2011.

R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *Proc. 2013 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '13)*, pages 13–24, 2013.

W. P. Yan and P.-A. Larson. Performing group-by before join [query processing]. In *Proc. 10th Int. Conf. on Data Engineering (ICDE '94)*, pages 89–100, 1994.

W. P. Yan and P.-A. Larson. Eager Aggregation and Lazy Aggregation. In *Proc. 21st Int. Conf. on Very Large Data Bases (VLDB '95)*, pages 345–357, 1995.

L. Yi, B. Liu, and X. Li. Eliminating Noisy Information in Web Pages for Data Mining. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 296–305, 2003.

Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 1–14, 2008.

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proc. 2nd USENIX Conf. on Hot Topics in Cloud Computing (HotCloud '10)*, pages 10–10, 2010.

J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. SCOPE: Parallel Databases Meet MapReduce. *VLDB J.*, 21(5):611–636, 2012.

H. Zou, Y. Yu, W. Tang, and H. M. Chen. Improving I/O Performance with Adaptive Data Compression for Big Data Applications. In *Proc. 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 1228–1237, 2014.

Appendix

1 Sopremo operators for IE and WA

This appendix lists all elementary and complex Sopremo operators and their instantiations. For each elementary operator, implemented algorithms, types, and prerequisites are stated (cf. Tables 1 and 2). For each complex operator, contained elementary components and prerequisites are listed in Table 4. Furthermore, all elementary and complex operators are described with properties relevant for semantics-aware data flow optimization (cf. Table 5). Each operator listed in the tables is assigned an ID starting either with "E" (for elementary operators) or "C" (for complex operators) followed by a numerical value. If more than one instantiation is available for an operator, a unique lower-case is appended to each operator ID. For example, the operator ID "E15a" identifies the instantiation `blp` of the boilerplate detection operator `dtct-bp`. If operator IDs are marked with a "*" symbol when describing a property, this property applies to all available operator instantiations. Consider Line 7 of Table 2. The operator instantiation "E4b" of `splt-txt` lists operator instantiations "E2*" as its prerequisite, i.e., any instantiation of operator "E2" (`anntt-tok`) cannot be reordered with E4b.

ID	Operator	Algorithm	Type	Prerequisites
E15a	`dtct-bp`	Boilerpipe	blp	–
E15b	`dtct-bp`	Boilerpipe	blp_largest	–
E15c	`dtct-bp`	Boilerpipe	news	–
E15d	`dtct-bp`	Cuter	–	–
E15e	`dtct-bp`	Snack	news	–
E15f	`dtct-bp`	RDB	–	–
E16	`rpr-mrk`	HTMLCleaner	–	
E17	`rm-mrk`	own	–	E15*
E18a	`dtct-struct`	own	table	–
E18b	`dtct-struct`	own	link	–
E18c	`dtct-struct`	own	list	–

Table 1: Elementary Sopremo operator instantiations for processing HTML documents (Sopremo web analytics package). Top: boilerplate detection and removal, bottom: structure detection.

ID	Operator	Algorithm	Type	Prerequisites
E1	`anntt-sent`	OpenNLP	–	–
E2a	`anntt-tok`	OpenNLP	sentence	E1
E2b	`anntt-tok`	OpenNLP	document	–
E3a	`anntt-ngram`	own	token	E2
E3b	`anntt-ngram`	own	character	–
E4a	`splt-txt`	own	sentence	E1
E4b	`splt-txt`	own	token	E2*
E4c	`splt-txt`	own	ngram	E3*
E5a	`anntt-pos`	OpenNLP	–	E1,E2a
E5b	`anntt-pos`	Medpost	–	E1,E2a
E5c	`anntt-pos`	TreeTagger	–	E1,E2a
E6	`anntt-stem`	OpenNLP	–	E1,E2a
E7a	`anntt-stop`	own	english	E2*
E7b	`anntt-stop`	own	biomedical	E2*
E8	`anntt-struct`	OpenNLP (Parser)	–	E1,E2a
E9	`repl-stem`	own	–	E6
E10	`rm-stop-anntt`	own	–	E7*
E11a	`anntt-ent`	OpenNLP	person	E1,E2a
E11b	`anntt-ent`	OpenNLP	date	E1,E2a
E11c	`anntt-ent`	OpenNLP	location	E1,E2a
E11d	`anntt-ent`	OpenNLP	money	E1,E2a
E11e	`anntt-ent`	OpenNLP	organization	E1,E2a
E11f	`anntt-ent`	OpenNLP	percentage	E1,E2a
E11g	`anntt-ent`	OpenNLP	time	E1,E2a
E11h	`anntt-ent`	Linnaeus	cell	E1,E2a
E11i	`anntt-ent`	Linnaeus	compound	E1,E2a
E11j	`anntt-ent`	Linnaeus	disease	E1,E2a
E11k	`anntt-ent`	Linnaeus	drug	E1,E2a
E11l	`anntt-ent`	Linnaeus	enzyme	E1,E2a
E11m	`anntt-ent`	Linnaeus	gene	E1,E2a
E11n	`anntt-ent`	Linnaeus	species	E1,E2a
E11o	`anntt-ent`	Linnaeus	tissue	E1,E2a
E11p	`anntt-ent`	Banner	gene	E1
E11q	`anntt-ent`	Geneview	disease	E1
E11r	`anntt-ent`	ChemSpot	drug	E1
E11s	`anntt-ent`	own	regex	E1,E2a
E11t	`anntt-ent`	own	exact	E1,E2a
E12	`anntt-rel`	own	co-occurrences	E1,E2a, E7*,E11*
E13	`mrg`	own		–
E14a	`emit`	own	entity	E11*
E14b	`emit`	own	relation	E12

Table 2: Elementary Sopremo operator instantiations for information extraction with associated algorithms, types, and prerequisites. Top: Text segmentation, middle: linguistic analysis, bottom: entity and relationship detection.

Algorithm	Source URL	Reference
OpenNLP	http://www.opennlp.org	
Medpost	http://sourceforge.net/projects/medpost/	[Smith et al., 2004]
TreeTagger	http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/	[Schmid, 1995]
Linnaeus	http://linnaeus.sourceforge.net/	[Gerner et al., 2010]
Banner	http://banner.sourceforge.net	[Leaman and Gonzalez, 2008]
Geneview	http://bc3.informatik.hu-berlin.de	[Thomas et al., 2012]
ChemSpot	http://github.com/rockt/ChemSpot	[Rocktäschel et al., 2012]
Boilerpipe	http://github.com/kohlschutter/boilerpipe	[Kohlschütter et al., 2010]
Cuter		[Krause, 2012]
Snack	http://github.com/karussell/snacktory	
RDB	http://www.readability.com	
HTMLCleaner	htmlcleaner.sourceforge.net	

Table 3: References and source code URLs for Sopremo operator algorithms.

ID	Operator	Elementary components	Prerequisites
C1	`splt-sent`	E1, E4a	–
C2a	`splt-tok`	E2a, E4b	E1
C2b	`splt-tok`	E2b, E4b	–
C3a	`splt-ngram`	E3a, E4c	E1
C3b	`splt-ngram`	E3b, E4c	–
C4a	`rm-stop`	E7a, E10	E2*
C4b	`rm-stop`	E7b, E10	E2*
C5	`stem`	E6,E9	E1,E2a
C6a	`extr-ent`	E11a,E14a	E1,E2a
C6b	`extr-ent`	E11b,E14a	E1,E2a
C6c	`extr-ent`	E11c,E14a	E1,E2a
C6d	`extr-ent`	E11d,E14a	E1,E2a
C6e	`extr-ent`	E11e,E14a	E1,E2a
C6f	`extr-ent`	E11f,E14a	E1,E2a
C6g	`extr-ent`	E11g,E14a	E1,E2a
C6h	`extr-ent`	E11h,E14a	E1,E2a
C6i	`extr-ent`	E11i,E14a	E1,E2a
C6j	`extr-ent`	E11j,E14a	E1,E2a
C6k	`extr-ent`	E11k,E14a	E1,E2a
C6l	`extr-ent`	E11l,E14a	E1,E2a
C6m	`extr-ent`	E11m,E14a	E1,E2a
C6n	`extr-ent`	E11n,E14a	E1,E2a
C6o	`extr-ent`	E11o,E14a	E1,E2a
C6p	`extr-ent`	E11p,E14a	E1
C6q	`extr-ent`	E11q,E14a	E1
C6r	`extr-ent`	E11r,E14a	E1
C6s	`extr-ent`	E11s,E14a	E1,E2a
C6t	`extr-ent`	E11t,E14a	E1,E2a
C7	`extr-rel`	E12,E14b	E1,E2a,E7*,E11*
C8	`rm-bp`	E16,E15*,E17	–

Table 4: Complex Sopremo operator instantiations for information extraction and web analytics with contained elementary components and prerequisites. First group: Text segmentation, second group: linguistic analysis, third group: entity and relationship extraction, fourth group: boilerplate detection.

ID	PACT(s)	I/O ratio	Record size	Schema handling	Processing type	Idem-potent	Commuta-tive with
E1	Map	$I = O$	$I \leq O$	extension	RAAT		E1
E2*	Map	$I = O$	$I \leq O$	extension	RAAT		E2*,E3*
E3*	Map	$I = O$	$I \leq O$	extension	RAAT		E2*,E3*
E4a	Map	$I \leq O$	$I \geq O$	modification	RAAT	✓	E11*, E12
E4b,c	Map	$I \leq O$	$I \geq O$	modification	RAAT		
E5*	Map	$I = O$	$I \leq O$	extension	RAAT		E5*, E6, E7*, E11
E6	Map	$I = O$	$I \leq O$	extension	RAAT		E5*, E6, E7*, E8, E11*, E12
E7*	Map	$I = O$	$I \leq O$	extension	RAAT		E5*, E6, E7*, E8, E11*, E12
E8	Map	$I = O$	$I \leq O$	extension	RAAT		E6,E7*, E11
E9	Map	$I = O$	$I \approx O$	unchanged	RAAT	✓	
E10	Map	$I = O$	$I \geq O$	unchanged	RAAT	✓	
E11	Map	$I = O$	$I \leq O$	extension	RAAT		E4a, E5, E6, E7*, E8, E11*
E12	Map	$I = O$	$I \leq O$	unchanged	RAAT		E4a, E6, E7, E12
E13	Match	$I = O$	$I \geq O$	modification	RAAT		
E14	Map	$I = O$	$I \geq O$	modification	RAAT		
E15	Map	$I = O$	$I \leq O$	extension	RAAT		E15*
E16	Map	$I = O$	$I \approx O$	unchanged	RAAT	✓	
E17	Map	$I = O$	$I \geq O$	extension	RAAT		
E18	Map	$I = O$	$I \leq O$	extension	RAAT		E18
C1	2x Map	$I \leq O$	$I \geq O$	modification	RAAT	✓	
C2	2x Map	$I \leq O$	$I \geq O$	modification	RAAT	✓	
C3	2x Map	$I \leq O$	$I \geq O$	modification	RAAT	✓	
C4	2x Map	$I = O$	$I \geq O$	unchanged	RAAT	✓	
C5	2x Map	$I = O$	$I \approx O$	unchanged	RAAT	✓	
C6	2x Map	$I \neq O$	$I \leq O$	modification	RAAT		
C7	2x Map	$I \neq O$	$I \leq O$	modification	RAAT		
C8	3x Map	$I = O$	$I \geq O$	unchanged	RAAT		

Table 5: Properties of elementary (top) and complex (bottom) operators for IE and WA.

2 Rewrite Rules

This section lists and explains the usefulness of all rewrite rules available in SOFA and based on the following scenario. Assume we have three datasets R, S, and T, where R consists of a set of unstructured texts together with IE annotations, and S and T hold information on income and assigned department of a set of employees. The JSON data formats for R, S, and T are shown in Listing 1 and in Listing 2, respectively. We exemplify the usage of our rewrite rules by applying different small data flows to R, S, and T.

The first two rewrite rules (cf. Listing 3 and 4) cover operator reorderings based on read/write set analysis similar to Hueske et al. [[2012]]. In contrast to Hueske et al. [[2012]], we define the rewrite rules not based on the type of the parallelization function used for operator implementation (e.g., map, reduce). Instead, we use the more general properties *processing type* and *number of inputs*, which also allows us to reorder complex operators exhibiting these properties.

Listing 1: Json format of dataset R.

```
{
  'id' : integer,
  'text' : string,
  'author' : string,
  'annotation' : {
    'linguistic' : {
      'sentence' : array,
      'token' : array,
      'pos' : array
    },
    'entity' : [{
      'id' : string,
      'text' : string,
      'type' : string
    }]
  },
  'year': integer
}
```

Listing 2: Json format of datasets S and T.

```
{
  'id' : integer,
  'name': string,
  'income' : double,
  'department' : string
}
```

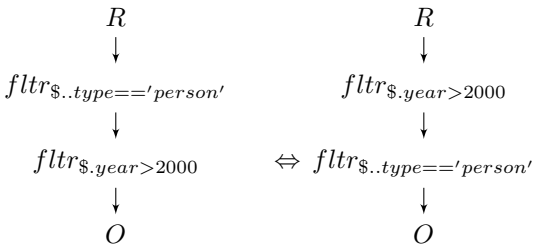
Specifically, the rewrite rule shown in Listing 3 states that two record-at-a-time, single-input operators can be reordered if they have no read/write conflicts. This allows us for example to reorder the two filter operators contained in the following data flow:

Listing 3: Rewrite rule 1.

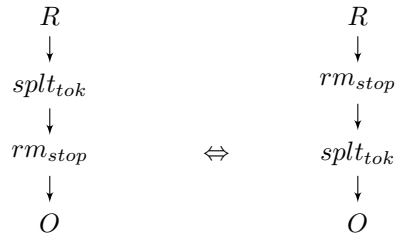
```
reorder(X,Y) :- hasProperty(X,'single-in'), hasProperty(X,'RAAT'),
                hasProperty(Y,'single-in'), hasProperty(Y,'RAAT'),
                noReadWriteConflicts(X,Y).

noReadWriteConflicts(X,Y) :- intersection(writeSet(X),readSet(Y),N),
                             intersection(readSet(X),writeSet(Y),M),
                             length(N,0), length(M,0).
```

Listing 4: Rewrite rule 2.

```
reorder(X,Y) :- hasProperty(X,'single-in'), hasProperty(X,'BAAT'),
                hasProperty(Y,'single-in'), hasProperty(Y,'RAAT'),
                noReadWriteConflicts(X,Y), preservesKeyGroup(Y).

reorder(X,Y) :- hasProperty(X,'single-in'), hasProperty(X,'RAAT'),
                hasProperty(Y,'single-in'), hasProperty(Y,'BAAT'),
                noReadWriteConflicts(X,Y), preservesKeyGroup(X).
```



It also allows us to rewrite a data flow that uses two complex operators to split text contained in the attribute $\$.text$ of items in dataset R into tokens and to remove stop-words:

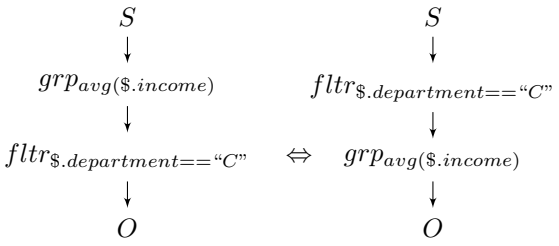


The second rewrite rule (see Listing 4) allows us to reorder bag-at-a-time with record-at-a-time operators similar to Hueske et al. [[2012]] given that both operators have no read/write conflicts and the record-at-a-time operator preserves key groups, i.e., retains or removes all items belonging to the same key group processed with the bag-at-a-time operator. As reordering based on read/write set analysis is symmetrical, the second rewrite rule consists of two parts to enable partial data flows $X \rightarrow Y$ where either X is the bag-at-a-time and Y the record-at-a-time operator or vice versa.

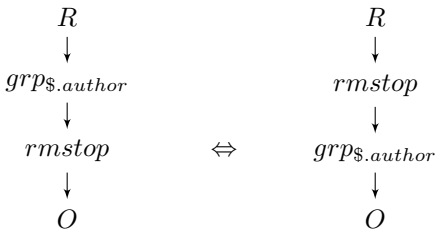
Listing 5: Rewrite rule 3.

```
1 reorder(X,Y) :- not hasPrerequisite(Y,X), isA(X,'annotate'), isA(Y,'annotate').
```

For example, a data flow that analyzes dataset S to determine the average income of people in department “C" can be reordered as follows:

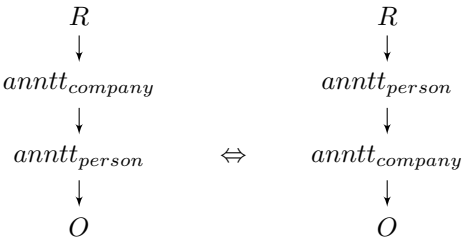


Analog to rule 1 and extending the read/write-set analysis as proposed in Hueske et al. [[2012]], we are can rewrite single-input bag-at-a-time operators with complex single-input record-at-a-time operators using the second rewrite rule, if no read/write conflicts exist and the complex record-at-a-time operator either removes or keeps all items belonging to the same key group. Consider a data flow that groups articles from data set R by author and removes stop words from the texts using the complex, single-input, record-at-a-time operator `rmstop`. This data flow can be rewritten as follows:



The third rewrite rule (see Listing 5) is specific to annotation operators as contained in the IE package. It enables reordering of any annotation operators X, Y given that X is not a prerequisite of Y. This rewrite rule is most useful for IE and NLP data flows containing many annotation operators.

For example, a data flow that annotates person and company names in items from dataset R can be reordered despite read/write conflicts (i.e., both `anntt` operators write to the attribute $\$annotation.entity$) in the following way:



Listing 6: Rewrite rule 4.

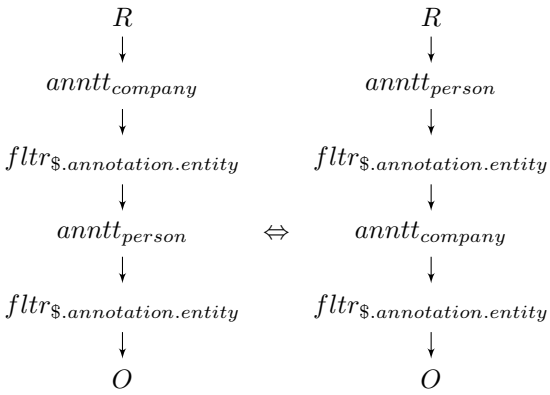
```
reorderAnnotateAndFilter(W,X,Y,Z) :- isA(W,'annotate'), isA(X,'filter'),
                                     isA(Y,'annotate'), isA(Z,'filter'),
                                     rwConflictsAnnotateFilter(W,X),
                                     rwConflictsAnnotateFilter(Y,Z).

rwConflictsAnnotateFilter(X,Y) :- not isEmpty(intersection(writeSet(X),readSet(Y))).
```

Listing 7: Rewrite rule 5.

```
reorder(X,Y) :- not hasPrerequisite(Y,X), isA(X,Z), reorder(Z,Y).
```

In IE data flows, we often observe that operators for entity or relation annotation are followed by `fltr` operators that keep only those data items containing entities or relations, respectively. To enable reordering of filter operators together with their corresponding annotation operators, we added a quite specific rule (see Listing 6) to our rule set.

Consider the following data flow, that annotates person and company names in the $\$.text$ attribute of dataset R. Each annotation operator is followed by a filter operator, which keeps only those items actually containing a person or company, respectively. Here, the goal `reorderAnnotateAndFilter` evaluates to true and thus, we allow to reorder the data flow as follows by removing the edge $fltr_{company} \rightarrow anntt_{person}$ from the corresponding precedence graph (see Section 5.3 for details):



The rewrite rule shown in Listing 7 enables reorderings of operators X based on inherited properties from an ancestor Z of X. Specifically, the rule states that if operator X is not a prerequisite of operator Y, X is a descendant of operator Z, and Z and Y are reorderable, then X and Y are also reorderable. This rule enables reordering of operators, which are not well annotated with property information, but which are declared as a specialization of some other operator.

Consider two operators `detect-link` and `detect-struct` from the WA package that detect outgoing links and structured information (e.g., tables, lists) contained in websites. Both operators save the link and structure information in certain attributes $\$.link$

Listing 8: Rewrite rule 6.

```
1 reorder(X,X) :- hasProperty(X,'associative'), isA('operator').
```

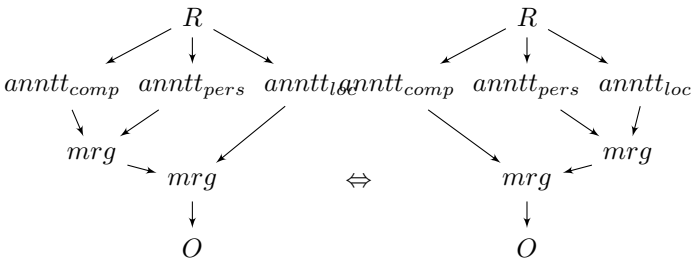
Listing 9: Rewrite rule 7.

```
1 removeX(X,Y):-X=Y, hasProperty(X,'idempotent').
```

and $.*structure*, respectively. For our example, we assume that both `detect-link` and `detect-struct` are not well annotated with properties, but are declared as specializations of the abstract IE annotation operator `anntt`. Now, all rules that are applicable to `anntt` (e.g., rule 4 and 5) become applicable to `detect-link` and `detect-struct`.

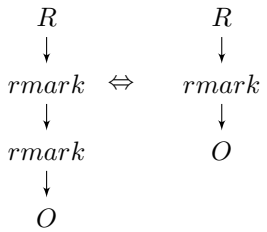
SOFA also employs rewrite rules based on algebraic operator properties for data flow rewriting, see Listings 8–11. Reordering of associative operators is covered in the sixth rule.

For example, a data flow containing two inner joins can be reordered in the same manner as in relational databases. Similarly, a data flow that performs task-parallel annotation of the three different entity types person, company, and location in dataset R and that subsequently merges these annotations can be rewritten, based on the associativity of the `mrg` operator:



Operator removals are possible if two adjacent operators of the same type and same instantiation are configured identically and furthermore annotated as idempotent (see Listing 9).

Assume that dataset R is a collection of HTML websites where all markup shall be removed from the attribute $.*text* of each input item. A data flow containing an operator `rmark` for markup removal twice can be rewritten:



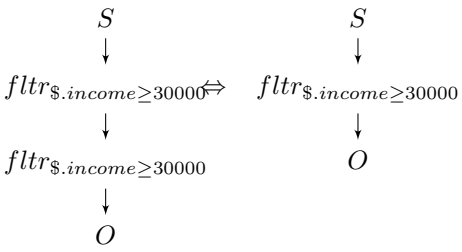
Similarly, two adjacent filter operators performing the same filter operation can be rewritten as follows:

Listing 10: Rewrite rule 8.

```
1 rewireInputs(X) :- hasProperty(X, 'commutative'), hasProperty(X,'dual-input').
```

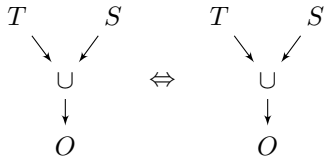
Listing 11: Rewrite rule 9.

```
1 rewriteDistributive(X,Y) :- hasProperty(X, 'distributive'), hasProperty(X,'BAAT'),
2                             hasProperty(Y,'RAAT'), hasProperty(X,'dual-input'),
3                             hasProperty(Y,'single-input').
```



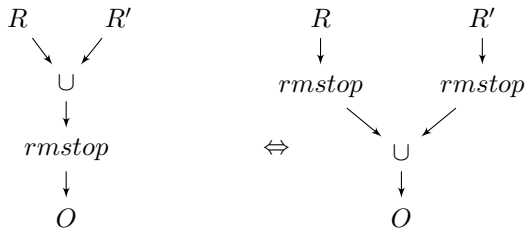
The rewrite rule shown in Listing 10 enables to interchange the left and right input of commutative dual input operators. In Stratosphere, the operators for unioning, intersecting, joining, and merging datasets are commutative.

Suppose that the datasets S and T shall be unioned and therefore, the following two data flows are equivalent:



Distributive dual-input bag-at-a-time operators can be rewritten with single-input record-at-a-time operators as indicated by the ninth rewrite rule of SOFA shown in Listing 11.

This rule applies to quite a few operator combinations, for example consider two datasets R and R' exhibiting the same schema that shall be unioned and afterwards, stop-words shall be removed from the attribute $\$.text$ by applying a regular-expression based stop-word removal operator `rmstop`. Such a data flow can be rewritten as follows:



Listing 12: Rewrite rule 10.

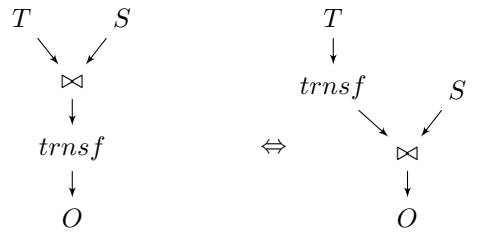
```
1 reorderWithLeft(X,Y) :- hasProperty(X,'dual-input'), hasProperty(Y,'single-input'),
2                         hasProperty(Y,'RAAT'), not contains(readSet(Y),readSet(X)),
3                         not contains(rightInputSchema(X),readSet(X)).
```

Listing 13: Rewrite rule 11.

```
1 removeX(X,Y) :- hasProperty(X,'RAAT'), hasProperty(Y,'RAAT'), hasProperty(X,'I=O'),
2                 hasProperty(X,'input-schema=output-schema'),
3                 not contains(readSet(Y),writeSet(X)),
4                 not contains(outSchema(Y),writeSet(X)).
```

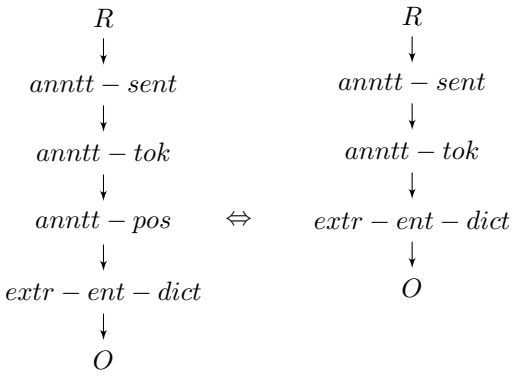
Rewrite rule 10 is again a very specific rule that allows to prepone a single-input record-at-a-time operator Y before a dual-input operator X given that Y only accesses attributes of the left input of Y. In this cases, if `reorderWithLeft` evaluates to true, Y can be placed directly before the left input of X (i.e., by removing the edge from the right input of X to Y and the edge $X \rightarrow Y$ from the corresponding precedence graph).

Suppose, we are given a data flow that consists of an equi-join of two datasets S , T followed by `trnsf` that transforms only attributes of S, which are not part of the join condition. This data flow can be rewritten into an equivalent data flow, which first applies `trnsf` to S and afterwards joins S and T:



Finally, the rewrite rule shown in Listing 13 enables to remove an operator X in case X has only one outgoing edge to some other operator Y and X modifies only attributes that are neither accessed by Y and not contained in the output schema of Y. In SOFA, plan optimizations involving operator removals is handled as follows: During precedence analysis, the rule for operator removal is evaluated before all other rules for plan transformation. If `removeX(X,Y)` evaluates to true for some operators X and Y, X and all incident edges of X are removed from the directed transitive closure D^+ of the given plan D. SOFA continues with analyzing precedence constraints in the newly formed graph and eventually enumerates plan alternatives as described in Section 5.3.

Consider an IE data flow that first annotates linguistic structure in the text (i.e., sentences, tokens, and part-of-speech tags) and afterwards extracts entities using a dictionary-based entity extractor. In this scenario, the `anntt-pos` operator can be removed since the annotations produced by `anntt-pos` are not read by the complex `extr-ent-dict` operator and are not contained in the output schema produced by `extr-ent-dict`:


R
annott − sent
annott − tok
annott − pos
extr − ent − dict
O
⇔
R
annott − sent
annott − tok
extr − ent − dict
O

3 Evaluation benchmark

Query 1 as shown in Listing 14 annotates relationships between drugs and genes on a collection of biomedical texts, which might contain duplicate texts. Therefore, duplicate removal is performed in advance as a preprocessing step.
Query 2 as shown in Listing 15 performs topic detection by computing term frequencies in a corpus grouped by year. The query first splits the input data into sentences, reduces terms to their stem, removes stopwords, splits the text into tokens, and aggregates the token counts by year.
Query 3 as shown in Listing 16 extracts NASDAQ-listed companies that went bankrupt between 2010 and 2012 from a subset of Wikipedia. This query takes article versions from two different points in time, annotates company names in both sets and applies different `fltr` operators and a `join` to accomplish the task.
Query 4 as shown in Listing 17 corresponds to the data flow shown in Figure 5.7 and performs task-parallel annotation of person and location names in Wikipedia articles created in 21013 or later.
Query 5 as shown in Listing 18 analyzes DBpedia to retrieve politicians named 'Bush' and their corresponding parties using a mixture of DC and base operators.
Query 6 as shown in Listing 19 is a relational query inspired by the TPC-H query 15. It filters the lineitem table for a time range, joins it with the supplier table, groups the result by join key, and aggregates the total revenue to compute the final result.
Query 7 as shown in Listing 20 uses two complex and two elementary IE operators to split incoming texts into sentences and to extract person names.

Listing 14: Query 1.

```
using base,ie,cleansing;

$article = read from "hdfs://192.168.127.43:50040/medline/";
$article = rdup $article;

$article = annotate sentences $article;
$article = annotate tokens $article;
$article = annotate postags $article;

$article = annotate entities $article use algorithm 'regex' type 'drug';
$article = filter $article where $article.annotation_entity.drugs;

$article = annotate entities $article use algorithm 'regex' type 'gene';
$article = filter $article where $article.annotation_entity.genes;

$article = annotate relations $article;
$article = filter $article where $article.annotation_relation;

write $article to "hdfs://192.168.127.43:50040/results/q1/";
```

Listing 15: Query 2.

```
using base,ie;

$input = read from 'hdfs://192.168.127.43:50040/medline/';
$input = split sentences $input;
$input = annotate tokens $input;
```

```
6
7  $input = annotate stems $input;
8  $input = replace with stems $input;
9
10 $input = remove stopwords $input;
11 $input = split text $input on "TOKENS";
12
13 $input = group $input by { $input.text, $input.year }
14            into {
15              text:max($input.text),
16              year:max($input.year),
17              wordcount: count($input.id)
18            };
19
20 $input = filter $i in $input where length($i.text)>3;
21
22 write $input to 'hdfs://192.168.127.43:50040/results/q2/';
```

Listing 16: Query 3.

```
1  using base,ie;
2
3  $new = read from 'hdfs://192.168.127.43:50040/wiki2012/';
4
5  $new = annotate sentences $new;
6  $new = annotate tokens $new;
7  $new = annotate entities $new type "organization";
8
9  $new = filter $article in $new where $article.annotation_entity.organization;
10
11 $new = filter $article in $new where (strpos($article.text,"bankrupt")>=0);
12
13 $old = read from 'hdfs://192.168.127.43:50040/wiki2010/';
14
15 $old = annotate sentences $old_articles;
16 $old = annotate token $old_articles;
17 $old = annotate entities $old_articles type "organization";
18
19 $old = filter $article in $old where $article.annotation_entity.organization;
20
21 $old = filter $article in $old where (strpos($article.text,"NASDAQ")>=0);
22
23 $old = filter $article in $old where (strpos($article.text,"bankrupt")<0);
24
25 $results = join $new, $old where ($new.id==$old.id);
26
27 write $results to 'hdfs://192.168.127.43:50040/results/q3/';
```

Listing 17: Query 4.

```
1  using base,ie;
2
3  $articles = read from 'hdfs://192.168.127.43:50040/wikipedia/';
4  $articles = annotate sentences $articles;
5  $articles = annotate tokens $articles;
6  $location = annotate entities $articles type "location";
7  $person = annotate entities $articles type "person";
8  $joined = merge $location, $person where ($location.id == $person.id);
```

```

$result = filter $article in $joined where ($article.year >= "2013");
write $result to 'hdfs://192.168.127.43:50040/results/q4/';
```

Listing 18: Query 5.

```
using base,cleansing;

$input = read csv from 'hdfs://192.168.127.43:50040/dbpedia/'
           columns ['subject', 'predicate', 'object']
           delimiter ' '
           encoding 'iso-8859-1'
           quote true;

$parties = filter $input where $input.predicate == '<http://dbpedia.org/ontology/party>';

$names = filter $input where $input.predicate == '<http://xmlns.com/foaf/0.1/name>';

$politicianWithName = join $p in $parties, $n in $names where $p.subject == $n.subject
           into {
             url: $p.subject,
             name: substring($n.object, 0, -3),
             party: $p.object
           };

$scrubbed = scrub $politicianWithName with rules {name: &normalizeUnicode};

$filtered = filter $p in $scrubbed where like($p.name, "%%Bush");

write $filtered to 'hdfs://192.168.127.43:50040/results/q5/';
```

Listing 19: Query 6.

```
$li = read from 'hdfs://192.168.127.43:50040/tpc-h/lineitem/';
$s = read from 'hdfs://192.168.127.43:50040/tpc-h/supplier/';

$join = join $li, $s where $s.s_suppkey == $li.l_suppkey
           into {
             $s.s_suppkey,
             $s.s_name,
             $s.s_address,
             $s.s_phone,
             $li.l_shipdate,
             $li.l_extendedprice,
             $li.l_discount
           };

$fli = filter $join where
           ($join.l_shipdate >= '1996-01-01' and $join.l_shipdate < '1996-04-01');

$revenue = group $fli by $fli.s_suppkey
           into {
             supplier_no: $fli[0].s_suppkey,
             total_revenue: sum($fli[*].l_extendedprice * (1-$fli[*].l_discount))
           };

write $revenue to 'hdfs://192.168.127.43:50040/results/q6/';
```

Listing 20: Query 7.

```
using ie;

$input = read from 'hdfs://192.168.127.43:50040/wikipedia/';

$input = split sentences $input;

$input = annotate tokens $input;

$input = annotate postags $input;

$results = extract entities $input type "person";

write $results to 'hdfs://192.168.127.43:50040/results/q7/';
```

4 Web-scale IE

Pronoun classes

Figure 1 shows the incidence of six different pronoun classes per document across the evaluated data sets.

Quality of boilerplate detection algorithms

Algorithms for boilerplate detection analyze the DOM tree of HTML documents with the goal to distinguish between relevant and irrelevant content (e.g., advertisement, navigational elements, user comments). In this section, we evaluate different approaches for boilerplate detection regarding extraction accuracy on three different gold standards.

The evaluated algorithms can be distinguished into two categories. The most common approach is to analyze HTML markup and the DOM structure of HTML documents, which is implemented by many open-source libraries as well as by commercial APIs. The second approach analyzes the rendered layout of web pages by decomposing it into different areas. Each area is subsequently classified depending on its relative position on the screen and each element of the underlying DOM-tree, which belongs to the area, is marked with the corresponding classification. We evaluated the algorithms `boilerpipe`[43], `krs`, `rdb`[44], and `snack`[45], which all analyze the DOM structure of HTML documents.

Data sets

We evaluated all algorithms on three different gold standard data sets, namely GoogleNews, CleanEval, and News600.

The GoogleNews data sets consists of 621 manually assessed HTML files, which were taken by random sampling from a crawl of approximately 250,000 news articles in 2008. The data set was created and annotated by Kohlschütter et al. [[2010]].

The CleanEval data set consists of 681 files from various domains and has been created and annotated by the ACL web-as-corpus community[46].

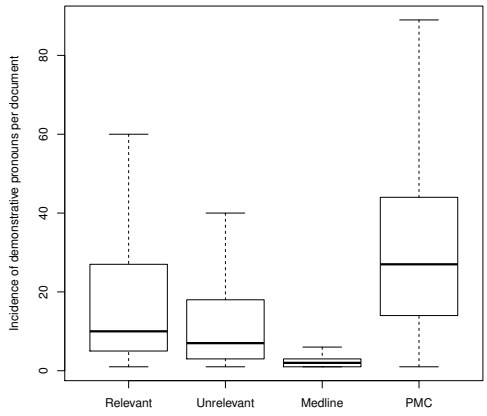
The News600 data set consists of 604 HTML files taken from news websites and has been created and manually annotated at the DOM tree level with 38 semantic labels by members of the Laboratoire d'Informatique de Paris 6, FranceSpengler and Gallinari [[2009]].

Evaluation

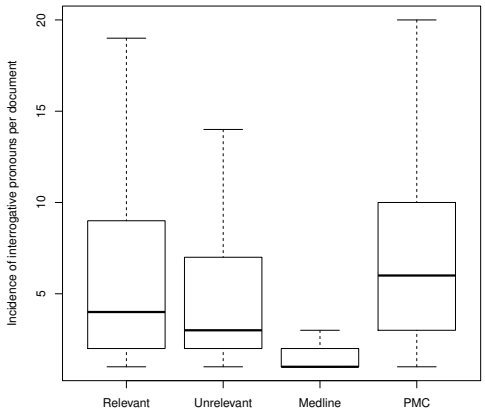
As shown in Figure 2, `blpLargest` achieved the highest precision on all data sets, followed by `snack`, which achieved the same scores for GoogleNews and CleanEval and a slightly lower score for the News600 data set. Concerning recall (cf. Figure 3), `blpNews`, `blp`, and `krs` scored best on the GoogleNews data set, whereas `blp` and `krs` scored best

[43] `https://github.com/kohlschutter/boilerpipe` (last accessed: 2016-10-05)
[44] `https://github.com/ifesdjeen/jReadability` (last accessed: 2016-10-05)
[45] `https://github.com/karussell/snacktory` (last accessed: 2016-10-05)
[46] `http://cleaneval.sigwac.org.uk/` (last accessed: 2016-10-05)

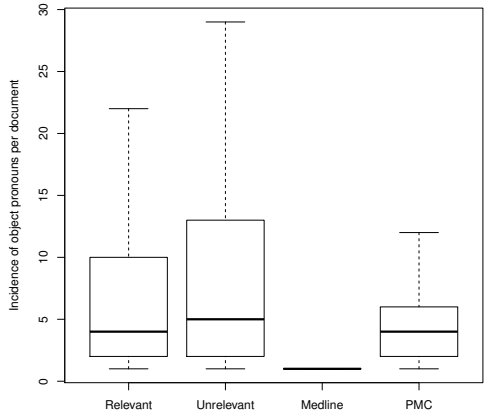
on CleanEval and News600. Interestingly, all algorithms only achieved moderate precision and recall scores on the CleanEval data set compared to the extraction results on the other data sets. We analyzed the types of errors made by the different tools (cf. Figure 4) and found that `krs`, `snack`, and `rdb` often extract no content at all. In contrast, `boilerpipe` extracts content from almost every tested web page and seems to be robust regarding diverse HTML files.



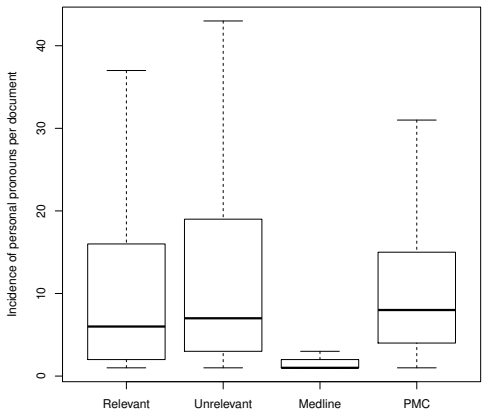
(a) Demonstrative pronouns



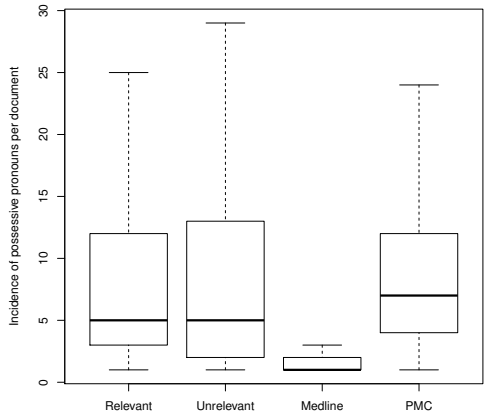
(b) Interrogative pronouns



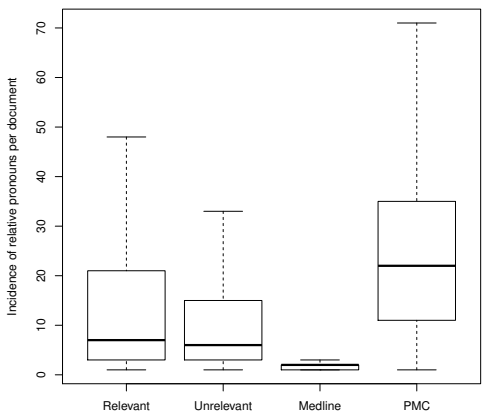
(c) Object pronouns



(d) Personal pronouns



(e) Possessive pronouns



(f) Relative pronouns

Figure 1: Incidence of six pronoun classes per document across different data sets.

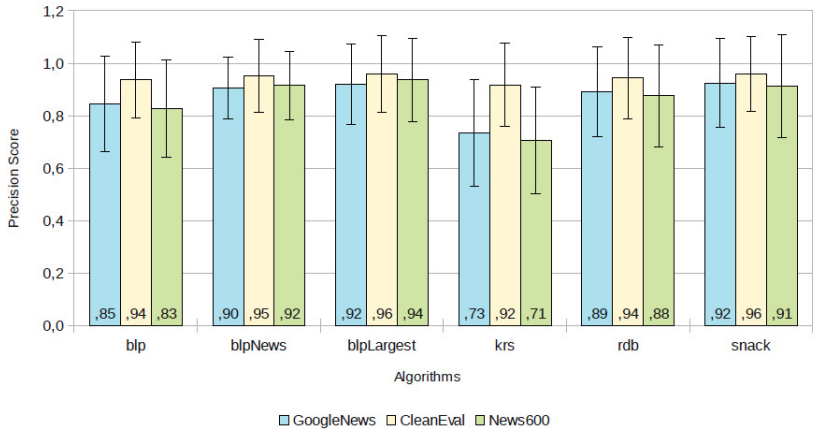


Figure 2: Precision of all boilerplate detection algorithms on all data sets.

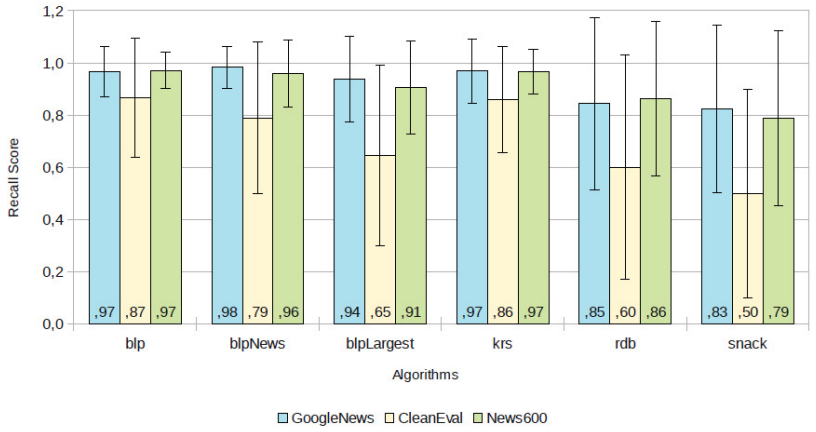
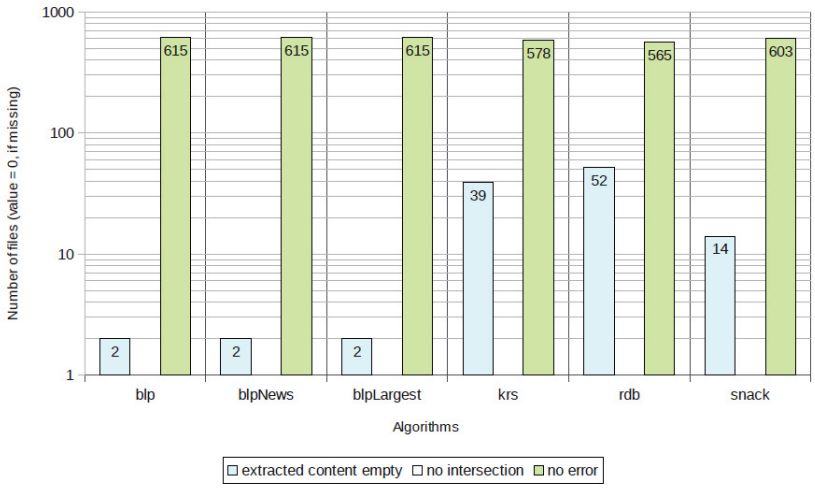
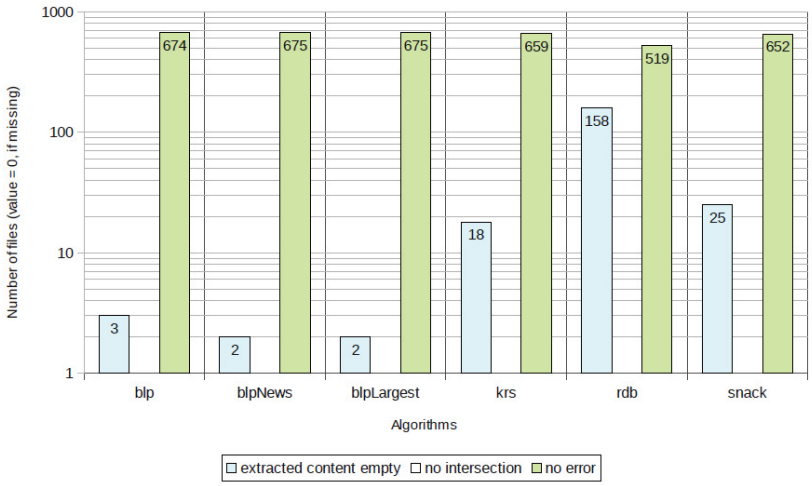


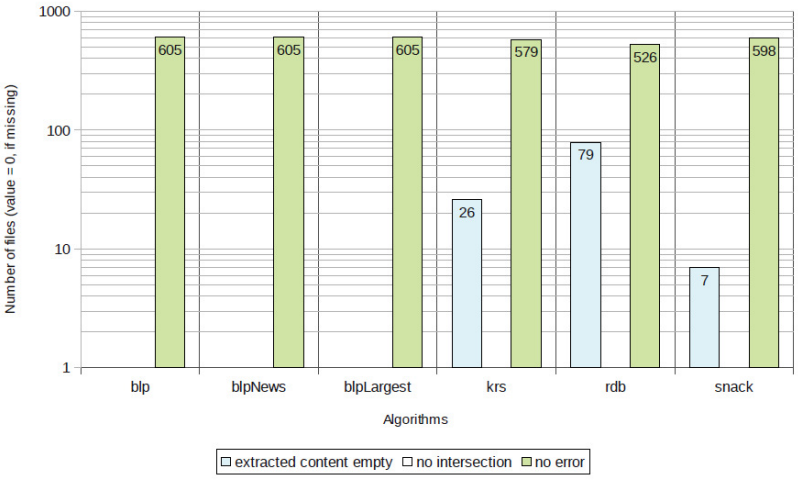
Figure 3: Recall of all boilerplate detection algorithms on all data sets.



(a) GoogleNews



(b) CleanEval



(c) News 600

Figure 4: Frequency and types of errors for all algorithms by data set.

List of Figures

List of Tables

Listings

Erklärung

Ich erkläre, dass ich die Dissertation selbständig und nur unter Verwendung der von mir gemäß § 7 Abs. 3 der Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 126/2014 am 18.11.2014, angegebenen Hilfsmittel angefertigt habe.

Berlin, den 24.01.2017 Astrid Rheinländer